1995

On the execution of high level formal specifications

Timothy Allen Wahls
Iowa State University

On the execution of high level formal specifications

by

Timothy Allen Wahls

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Department: Computer Science
Major: Computer Science

Approved:

Signature was redacted for privacy.

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Members of the Committee:

Signature was redacted for privacy.

Signature was redacted for privacy.

Signature was redacted for privacy.

Iowa State University
Ames, Iowa
1995

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

Thanks to my parents, Allen and Ellen Wahls, for their continuing support of my academic career. Thanks to my major professors, Albert Baker and Gary Leavens, for encouragement and inspiration. Thanks to my fiancee, Linda Null, for putting up with me while I was writing and grumbling. And a special thanks to IBM Rochester for the fellowship that supported me through much of the research that appears in this dissertation.

1. GENERAL INTRODUCTION

1.1 Problem Statement

While the formal specification of software systems has generated volumes of research and is commonly taught in undergraduate and graduate courses, formal specifications are not widely used in industrial software development. The advantages of formal specifications are clear: they precisely describe the functionality, structure, and interfaces of software systems without involving the programming language details needed to produce an implementation. The specifier works at a higher level of abstraction than the programmer, and so has the opportunity to produce a concise definition of system functionality without worrying about algorithms, efficiency, memory management, and other aspects of implementations that have nothing to do with the functional behavior of the system. Abstracting away these details decreases the likelihood of errors in the specification and allows readers of the specification to perceive the defined functionality without the confusion and verboseness that such details bring. This allows the correct use of an implementation of a formal specification without reading the code of that implementation. Formal specifications also permit the verification of implementations. For safety critical applications, proof that the implementation satisfies the specification may be vital. Thus, the academic interest in formal specifications is not surprising.

On the other hand, a number of factors have kept formal specifications from wide acceptance in industrial settings. Many software developers are not familiar with the notations used in formal specifications, and training in writing, reading, and implementing specifications is expensive and time consuming. Compounding this problem, programmers often consider implementations to be the only important product of software development, and so often have little enthusiasm for even infor-

mal specification and documentation. As specifications are written by humans, they often contain errors – specification of unintended and undesirable behavior. These errors can be found by examination of the specification (possibly including unsuccessful attempts to prove desirable properties of the specification), or by producing and testing an implementation of the specification. Examining and proving properties of specifications is difficult and labor intensive, even with the help of software tools, while producing an implementation of an erroneous specification wastes development resources. Additionally, formal specification activities produce little that can be demonstrated to clients as progress toward a completed system, as clients are unlikely to know the specification language and so are unable to read the specification. Unfortunately, this also means that a formal specification usually does not serve as a basis for discussion of whether the specified functionality is in fact the functionality desired by the client. Formal verification of implementations is tedious, error-prone, and labor intensive, so that such verification is rarely done even when a formal specification exists. Formal specifications are of some help in validating implementations, as they are an unambiguous definition of the correct output for any test. However, the tester must manually verify that the result of each test is consistent with the specification, which is again labor intensive and error-prone.

Executable specifications may be the key to increasing the usefulness and acceptance of formal specifications in industrial settings. Executable specifications can be debugged in much the same way that programs are debugged, as such specifications can easily be tested on a variety of inputs. This allows early and inexpensive detection and correction of errors in the specification. An executable specification is a working prototype of the final system, so that formal specification activities can also be viewed as progress toward a prototype. The prototype thus produced can be demonstrated to the client both as progress toward a final system, and to allow discussion of the desired functionality of the system as compared to the functionality expressed in the specification. A prototype is an excellent way to evaluate this functionality against the informal requirements of a client, as the client can directly determine whether the functionality satisfies the requirements. An executable specification helps in breaking down the barrier between specifiers and others involved in the software development process created by the formality of the specification language. While executable spec-

ifications provide no help in verifying implementations, they can serve as test oracles and so help in the validation of implementations. The behavior of implementations can be tested simply by running the implementation and executable specification on the same data and comparing the results. This allows considerable automation of the testing process and lessens the chance of human error.

Many executable specification languages have been developed, and many different techniques have been used for executing specifications. Some of the relevant research is described in the literature review (Section 1.3). As the literature review will show, existing executable specification languages often force the specifier to work at a low level of abstraction. This occurs because the specifier is forced to provide many of the implementation details in order to produce an executable specification. As an extreme example, some executable specification languages require the specifier to write parts of the specification in a production programming language. More commonly, a formal specification language is restricted to a subset that is algorithmic enough to allow easy execution. In either case, the expressiveness of the specification language is reduced, and the clarity and conciseness of non-executable specifications is compromised by the inclusion of detail needed only for achieving executability. As a specification language becomes more like a programming language, the specifier's task of producing a clear and concise definition of system functionality becomes more and more confounded by the details needed for execution.

Thus, executable specifications are desirable, but only if such specifications maintain the characteristics of non-executable specifications. This dissertation describes research into executing specifications written at a higher level of abstraction than that found in existing executable specification languages. The particular focus is on developing an execution technique for model-based specification languages, where model-based means that operations are specified via first order predicate logic pre- and post-conditions written over a fixed set of abstract model types. Typically, such model types include finite sets and sequences, tuples, reals, integers and so on. Although the execution technique was developed in the context of the model-based specification languages DFD-SPECS and SPECS-C++, and the description of the technique in the dissertation uses syntax and examples from these languages, the execution technique is not specific to these languages. The execution technique can

easily be adapted for any specification language that uses first order predicate logic assertions over a fixed set of model types. All that is required is to add code for handling any additional model types found in these languages. Examples of suitable languages include VDM [Jon90], Z [Hay87, Spi92, Spi88, Spi89], and the myriad dialects and object-oriented offspring of these languages. As VDM and Z are the dominant specification languages in both industrial usage and academia, the execution technique has the potential for much wider use than any interpreter or compiler that is tied closely to a particular language. Thus, the key contribution of the dissertation research is an algorithm for executing assertions over model types.

1.2 Dissertation Organization

This dissertation is organized as a collection of papers on executing formal, model-based specifications. The papers are preceded by a general introduction and literature review, and followed by a general summary. All references cited in the dissertation (including the collected papers) follow the general summary. The literature review is meant to be self-contained, and so is in part redundant with literature reviews appearing in the included papers. As the collected papers are also self-contained, similar motivational material appears in several of the papers. However, the technical content of each paper is different, as each paper focuses on a different aspect of formal specifications or executing formal specifications.

The first paper appearing in this dissertation, *A Case Study of Approaches to the Formal Specification of Abstract Data Types* (Chapter 2), could be considered part of the literature review, as it compares and contrasts the use of model-based, executable, and object-oriented specification languages in specifying ADTs. The second paper, *An Approach to the Execution of a Formalized Data Flow Diagram Specification Language* (Chapter 3), describes the development of an interpreter for DFD-SPECS, a formalized Data Flow Diagram specification language. During this research, it became clear that this first technique for executing the predicate logic assertions was inadequate for executing some interesting specifications, and that a general technique for executing assertions would be useful for executing any model-based specification language. This realization lead to the research described in the third paper, *An*

Approach to the Direct Execution of Model-Based Specifications (Chapter 4). This paper describes the first incarnation of the execution technique and its application in executing SPECS-C++, an interface specification language for specifying C++ classes. Later, this execution technique was applied to DFD-SPECS, resulting in the paper appearing in Chapter 3. Although this version of the execution technique was adequate for executing a large and useful subset of both DFD-SPECS and SPECS-C++, it did not handle underdetermined specifications well, and could not execute incomplete specifications. The author began investigating constraint logic programming languages as a possible means of correcting these deficiencies. This investigation lead to the final paper, *Applications of Constraint Logic Programming Techniques in Executing Specifications* (Chapter 5), which describes the refinement of the original execution technique using ideas from constraint logic programming. This paper includes a brief summary of the literature on constraint logic programming, and compares the refined execution technique with some of the major constraint logic programming languages.

1.3 Literature Review

The dominant themes of the executable specification literature are: specification languages that require the specifier to provide the algorithms needed for executing specifications and specification languages that rely on Prolog [CM84] for executability. This section presents examples of both themes, as well as a critical discussion of them. Particular attention is given to work on executing VDM [Jon90] and Z [Hay87, Spi88, Spi89, Spi92] specifications, as these languages are the dominant model-based specification languages in use today and work on executing these languages is directly comparable with our execution technique.

1.3.1 Algorithmic Specifications

As the name suggests, algorithmic specifications define operations and other parts of software systems by providing specific algorithms with the desired functionality. They are inherently easier to execute than specifications written at a higher level of abstraction, and so a great number of algorithmic specification languages

have been developed. This section reviews a sampling of these languages and then provides a critique of algorithmic specifications.

The aSet specification language [GRW91] with ACTESS modules provides a CASE (Computer Aided Software Engineering) tool for writing executable specifications based on Data Flow Diagrams (DFDs) [You89]. The ACTESS modules are used to define the functionality of bubbles. They are written in a procedural "Pascal-like" language, and so this is a clear example of an algorithmic specification language.

The Prosper specification language [LB89] [BY91] [BY92] [YB92] is designed for prototyping, and particularly for providing an oracle for automated testing of the finished system. Prosper also features an interesting dynamic and dependent type system. Prosper is classified as an algorithmic specification language because operations are specified in a functional "Lisp-like" programming language that provides the basis for the execution of Prosper specifications.

PAISley [ZS86] is an executable specification language based on asynchronous processes, where each process is much like a finite state machine. The computations that occur on state transitions are written in a functional programming language that features composition, conditional selection, and tuple formation.

PSDL [BL90] [LVY88] is an enhancement of DFDs for hard real-time specification. The language includes an interesting and elegant type system for flows. It qualifies as an algorithmic specific language because the atomic operations of the specified system must be written in some programming language.

While functional programming languages are seldom thought of as specification languages, Harry [Har92a] presents a number of features of functional languages that makes them suitable for producing prototypes. These features include their somewhat declarative nature, preservation of referential transparency (in other words, lack of side effects), strong typing, and resemblance to model-based specification languages. Peyton-Jones [PJ86] adds lazy evaluation, modularity and abstraction, and the possibility of automatic transformation to more efficient programs to this list. Some examples of, and techniques for, such transformations to iterative programs are presented in [FWH92]. This provides a basis for refinement of functional prototypes to more efficient, imperative implementations. Eisenbach and Sadler [ES90a] have combined the observations of Glaser et. al [GHT84] and Fairley [Fai85] to conclude

that functional programs can be produced an order of magnitude more quickly than imperative programs, thus adding to their suitability as prototyping languages.

The *me too* executable specification language [Hen86] is another example of the specification potential of functional languages, as its language constructs are taken from functional languages, notably Miranda [Tur85] and KRC. The development methodology of *me too* is largely taken from VDM. Specifications are executed by embedding them in Lisp.

The work of Johnson and Sanders [JS90] is also based on functional programming, as they describe a semi-formal technique for translating Z specifications into Lazy ML [AJ88]. Their work includes implementations of Z's built-in operators, although quantifiers appear to be restricted to ranging over finite lists of boolean values. They emphasize correctness preserving transformations to increase efficiency of prototypes, and the use of lazy evaluation to produce (possibly infinite) lists of solutions.

One prevalent subfield in algorithmic specifications is work on executing parts of the explicit subset of VDM. Explicit VDM specifications consist of the direct definition of pure functions, as opposed to implicit specifications that use first order predicate logic pre- and post-conditions to specify procedures and functions. Thus, explicit VDM specifications are somewhat similar to function definitions in pure functional programming languages, and researchers have exploited this similarity by using functional programming languages to execute parts of the explicit subset. Combining this with the fact that VDM is one of, if not the, most commonly used formal specification language, it is not surprising that a reasonably large body of work has grown around techniques for executing the explicit subset of VDM.

An early example of this work is the Meta IV compiler described in [Haß87]. (Meta IV is the formal name of the specification language associated with the Vienna Development Method, or VDM). The compiler described in this work is specialized to compiler generation – compiling VDM specifications of compilers to Pascal. It uses lazy evaluation, and so can handle infinite data structures. Another interesting feature is the use of executable fixed points for evaluating both lazy expressions (as they aren't directly translatable to Pascal) and simultaneous let expressions.

Another approach using lazy evaluation is that of [BM93], which provides an in-

formal algorithm and a semi-automated transformation for translating explicit VDM specifications to Lazy ML [AJ88]. The work includes a Lazy ML implementation of VDM's data types.

The IPTES system [AELL92] [LL91] is a methodology and environment for incremental prototyping of embedded systems. The sequential components of the system are specified in a subset of the explicit subset of VDM tailored toward this application. The system includes an interpreter for this subset. One interesting feature of this interpreter is that it allows incomplete specifications – specifications that are nondeterministic or underdetermined. For incomplete specifications, one may adopt a loose semantics, in which the specification determines a set of non-isomorphic models for the specified system. This allows the specifier to delay the choice of a specific model, or simply not limit the specification to one model, as appropriate. See [HJ89] for a discussion of the advantages of loose semantics. The approach taken in the IPTES system is that incomplete specifications are underdetermined, and so the system provides a single model that satisfies the specification. Hence, it does not use a loose semantics.

The UK's National Physical Laboratory has also been a major player in this area, as evidenced by its work on translating the explicit subset of VDM to both Miranda [Nor90] [O'N89] (a functional programming language somewhat similar to SML) and Standard ML [O'N92a] [O'N92b]. The contribution is the development of a syntax-directed editor for entering VDM specifications that can also translate the specification entered directly to SML code. The technique can even handle a (small) subset of implicit VDM specifications – those written in an "algorithmic enough" style for the translation to take place. As the editor allows any syntactically correct VDM specification to be entered, the resulting SML code must often be hand edited to account for limitations of the translation technique.

The EPROL specification language makes a subset of Meta IV executable by compiling it to Lisp. It is part of the EPROS system, which provides tool support for evolutionary and functionality prototyping and for building the user interface to the prototype. EPROL is especially interesting with respect to the execution technique described in the rest of this dissertation in that it also allows the execution of some implicit specifications. Unfortunately, the primary reference on EPROL [HI88]

does not carefully characterize the subset of VDM that EPROL makes executable. However, from the examples presented, it appears that a quantified expression can only be evaluated if it contains no references to post-state values (and so can only be evaluated for its boolean value) and that the only operator that can be used to define post-state values is equality. Set and list comprehensions are used to get around these limitations. Thus, EPROL specifications must provide a level of algorithmic detail similar to that found in modern functional programming languages. See [IWH86, Har92b] for other reviews of EPROL.

In summary, considerable research has gone into developing executable, algorithmic specification techniques. However, algorithmic techniques as a class are written at an inappropriate level of abstraction for specifications. Ideally, a specification should only define the functionality of a system. Inclusion of other information, such as the algorithms needed to actually produce the specified behavior, serves only to make the task of the specifier more difficult and to make readers of the specification work harder to understand the specified functionality. Thus, there is good reason to investigate techniques for executing non-algorithmic specifications.

1.3.2 Prolog-based Specifications

This category includes both specification languages whose syntax and semantics are based on Prolog, and also specification languages that provide executability by translating specifications into Prolog.

The PLEASE development system [TC89] incorporates an executable specification language with a distinctly Prolog-like syntax. Specifications are translated to Prolog for execution. Somewhat similarly, RSF [DFPT90], a language for real-time specification, uses Prolog syntax for expressions denoting both time stamps and values. More could be said about these languages, but their only relevance here is as examples of techniques based on Prolog.

Lazarev has advocated translating informal DFDs directly to Prolog [Laz89a] [Laz89b]. This technique cannot be automated, as it starts with an informal specification, but Lazarev provides an informal algorithm for making the translation. The user must also translate (completely by hand) the "Structured English" descriptions of bubble functionality to Prolog, as well as declaring the relationships between flows

in Prolog.

For purposes of this research, the most interesting work that has been done in this area is the use of Prolog for executing and animating Z specifications. West and Eaglestone [WE92] describe an approach called structure simulation for translating a subset of Z into Prolog. This includes rules for translating schemas and several of the schema calculus operators, as well as Prolog implementations of Z set and function operators. However, this approach is limited in that it only supports sets, tuples, and functions represented as sets of tuples. Additionally, only a limited form of existentially quantified expressions and no universally quantified expressions can be handled by their translation technique. Finally, hand editing of the Prolog code resulting from the translation is often required to re-order clauses within the bodies of rules so that they can be executed.

West and Eaglestone point out two significant ways in which their work differs from most other Z to Prolog approaches:

- They require the user to enter all test case sets as Prolog lists, while most approaches generate test cases from the translation of the schema signature. However, this generation approach produces an exponential number of test cases (in the size of the input), so some control of this process is required. For example, Dick et. al. [DKC90] describe this generate and test approach, and make their Prolog programs more efficient with a number of transformations, including replacing goals with more efficient (but equivalent) goals, and rearranging the order of goals in a rule.

- They use procedural operators (i.e., cut) in their Prolog implementations of Z operators, while most approaches stick to the pure subset of Prolog.

While Prolog has been applied in the research cited above, it does have a number of flaws when used as a specification/prototyping language. Languages that incorporate Prolog syntax necessarily expose readers and writers of specifications to that syntax. While Prolog syntax is a representation of a subset of first order predicate logic, it is very different from the syntax used in non-executable specification languages, and restricts the forms that logical assertions in specifications can take. Prolog programs also use relations in place of functions, which often confuses novice

Prolog users. Thus, Prolog is less expressive than first order predicate logic. Additionally, Prolog is not a pure logic programming language. Order of statements matters (as can be seen in West and Eaglestone's work, where hand re-ordering is required), and the language includes non-logical features such as ! (cut), fail, assert, and retract. These features are present largely to increase the efficiency of Prolog programs, and so have little meaning when Prolog programs are interpreted as functional specifications. Even Prolog programs that don't directly use these features (such as direct translations of formal specifications) often use not as logical negation, where not R is usually implemented as R, !, fail. Thus, it is difficult to stay within the logical interpretation of Prolog even when directly translating specifications. Both the loss of expressiveness with Prolog as compared to first order predicate logic and the use of non-logical features make Prolog programs considerably less abstract than non-executable model-based specifications.

Another serious flaw appears whenever specifications are translated to some programming language (so this charge applies to many of the techniques in the first section as well). When such translation is done, errors in the resulting prototype are reported by whatever the target language is. The translation system can help by generating code that gives helpful error messages, but this is difficult and it seems unlikely that all error conditions can be accounted for. For example, West and Eaglestone make no attempt to give such messages. Thus, the tester must know the target language well enough to determine what the error is, and also know the translation scheme well enough to invert it and find the cause of the error in the original specification. These are clearly nontrivial tasks, and so are burdens on users of specifications that are executed by such a translation.

So, there are faults with both algorithmic and Prolog-based approaches to executable specification, and good reasons to look for techniques that do not fit into either of these categories. The literature review concludes with a review of two such languages.

1.3.3 Other Approaches

While most executable specification languages fit neatly into either the algorithmic or Prolog-based categories, one language found fits equally well into both, and

one fits into neither.

The OBSERV prototyping language [TY92] models systems as collections of objects. The behavior of objects is modeled by finite state machines, with Prolog used to specify the activities within objects and the transitions between object states. However, the objects also have state variables, and these are modified using a "Pascal-like" language with assignment. Hence, OBSERV can be said to fit into both the Prolog-based and algorithmic categories, and so is open to most of the criticisms stated earlier.

The **fase3/C++** language of Kamin and Kraus [KK93, Kra88] is a (mostly) executable interface specification language for C++ classes. The **fase3** approach is very similar to that taken in the Larch [GHW85, GHG+93] family of specification languages, as it is two-tiered, consisting of the **fase3** shared language and the **fase3/C++** interface language for C++. The shared language is where most specification occurs, and so where interesting execution occurs as well. Only a few primitive types such as integer, boolean, and so on are built into **fase3**. More structured types, such as the set, sequence, and tuple types found in model-based specification languages are specified in the shared language using a unique style of algebraic specification that allows any type to be represented as a tuple of functions. As long as these functions remain finite for a particular element of the type, the functions can be represented as "tables" (finite sets of tuples), and quantified assertions over that element can be executed. The syntax for quantified assertions is elegant and concise, as the specifier need not supply an explicit bound on the quantified variable, as is required by other executable specification languages that allow explicit quantifiers. Assertions that use observers[1] other than equality to define values are also executable — in fact, the functions that represent an element are exactly the observers of the type as applied to that element.

In **fase3**, specifications are executed by first compiling them to an extended λ-calculus, and then to an extended form of combinator graph, which is then reduced. This execution strategy can execute many forms of quantified assertions. However, only one kind of **fase3** quantified assertion can be evaluated if its body contains refer-

[1] Observers are functions that, when applied to a value, return some aspect of that value.

ences to the function return value being defined. This kind of assertion is a restricted form of existential quantification that can only be used to find a particular value for the function result that satisfies the body of the quantified assertion. Universally and other existentially quantified assertions cannot refer to the function result. Thus, a large class of quantified assertions that are useful in writing specifications are not executable. The usefulness of errors reported by the reduction algorithm in debugging the specification is also questionable, as the combinator graph is two compilation steps removed from original specification.

1.3.4 Summary

Thus, all of the executable specification techniques reviewed suffer from at least one of the following problems:

- operations are specified algorithmically, and so at a low level of abstraction
- specifiers are forced to use the data types of executable languages like Lisp and Prolog.
- specifications are executed by translation to some programming language or other formalism, which complicates testing and debugging specifications
- specifications use Prolog syntax or are executed by translation to Prolog, and so expose the user to Prolog syntax and often to the non-logical features of the language

The problems associated with these techniques have already been highlighted.

So, the overall goal of the research described in this dissertation is to develop an execution technique for model-based specification languages that can execute specifications written at a high level of abstraction. The execution technique should not require users of executable specifications to know any programming or specification languages other than the language used for specifications, and it should be capable of reporting useful error messages in the context of the specification. The execution technique also should execute specifications written using the syntax of existing non-executable specification languages, and should allow the specifier to think only

of the logical interpretation of the specification language while writing executable specifications. The key to meeting these goals is a general algorithm for executing assertions.

2. A CASE STUDY OF APPROACHES TO THE FORMAL SPECIFICATION OF ABSTRACT DATA TYPES

A paper in preparation for Computing Surveys

Tim Wahls, Albert L. Baker, and Gary T. Leavens

Abstract

This paper compares five formal specification languages on the basis of specifications of the same two abstract data types (ADTs) in each language. The languages included are VDM, Z, SPECS-C++, Prosper, and Prolog. VDM and Z are the "industry standard" formal specification languages and so provide a known baseline for comparison. SPECS-C++ is an object-oriented interface specification language, and so represents both the growing number of object-oriented specification languages and interface specification languages. Prosper is an executable specification language wherein the necessary algorithms for execution are given in a functional programming language, and so typifies one approach to executable specifications. Many other executable specification languages either directly incorporate Prolog code in specifications or use Prolog as the engine for execution. The executable subset of SPECS-C++ is also compared with these executable specification languages. The specifications of the ADTs Table and Bounded List allow comparison of both highly practical and more theoretical differences between the specification languages. These comparisons provide a brief introduction to the languages and highlight some of the differences in language features and specification styles that impact reading and writing specifications.

2.1 Introduction

Model-based specification languages have dominated industrial use of formal methods, as evidenced by the popularity of VDM [Jon90] and Z [Hay87, Spi92, Spi88, Spi89]. Recently, many executable and object-oriented specification languages have been proposed, accompanied by claims that these approaches enhance the role of formal specifications in software development. In this paper, we compare and contrast these approaches by giving specifications of the same abstract data types in languages that represent each approach, and then comparing the difficulties of writing the specifications and the features of the resulting specifications.

The example ADTs, Table and Bounded List, were chosen because they are small enough to allow complete and at least somewhat realistic specifications to be included in this paper, because they typify the reusable abstraction that can often be cleanly separated from a larger system, and because we believe that they provide a fair basis for comparison. The specifications of ADT Table appear in Section 2.2 and of ADT Bounded List in Section 2.3. To assist readers who are unfamiliar with a particular language, some explaination of relevant language features appears with each specification. These specifications allow the languages to be compared on issues that would arise frequently in day-to-day use of a specification language. The example specifications follow the styles and conventions of each language, while keeping the models (for the model-based languages) and interfaces as similar as possible. When these goals conflict, language conventions are favored. Following these practical comparisons, the languages are compared from a more theoretical standpoint – on issues arising from the design philosophies behind the languages and on the intended applications of the languages.

The languages compared are VDM, Z, SPECS-C++ [Bak93], Prosper [LB89, LB88, BY91, BY92, YB92], and Prolog [CM84, Coh85]. VDM and Z are included in this comparison as the standard specification languages in use today, with the largest user communities and widest array of tools supporting specification activities.

SPECS-C++ is a recently developed object-oriented specification language which is tailored to specifying the interfaces of C++ [Str91] classes. Class and member function interfaces are given in C++ syntax, and a SPECS-C++ class specification must

be implemented by a C++ class with the same interface. The domains of SPECS-C++ are abstract model types similar to those of VDM and Z, except for the exclusion of function and relation types and the inclusion of object types. SPECS-C++ objects are similar to cells in imperative programming languages and locations in denotational semantics. Objects contain values, and the same object can have many names – for example, an object can appear as an element of a set and as an element of a sequence simultaneously. Any change in the contained value is visible from any occurence of the object. Thus, objects allow uniform detection and specification of mutation and aliasing, which is critical in writing practical specifications of C++ implementations.

The state of SPECS-C++ class instances is modeled by a tuple of abstract data members. For example, the abstract data members of class **BndList** (Section 2.3.3) are theList, currCursor, and maxSize. Abstract data members exist only to provide the abstract values of class instances, and so need not be implemented directly. However, for any correct implementation, there will be a mapping from the implementation data members to the abstract data members. In specification inheritance, a derived class inherits all of the abstract data members of its base classes. When a member function from a base class is respecified in a derived class, the base class specification is inherited. Thus, the implementation of the derived class member function must satisfy the base class specification's post-condition whenever the base class specification's pre-condition is satisfied. This kind of inheritance forces subclasses to be behavioral subtypes, and is similar to that found in Larch/C++ [LC93] and Eiffel [Mey90]. Other object-oriented specification languages include dialects of VDM and Z such as Fresco [Wil92], Object-Z [CDD+90] and Z++ [LH92], and the languages featured in [SBC92].

Prosper is an executable specification language that relies on functional programming language techniques for its executablity. Thus, it has much in common with *me too* [Hen86], EPROL [HI88, HI86], and *SMLVIEW* [O'N92a, O'N92b]. However, Prosper uses a powerful dependent type system not shared by any of these other languages. In all of these languages, the algorithms necessary for execution must be written by the specifier in a functional style, although the syntax involved varies from Lisp-like to a subset of VDM. Another variation on this theme is PAISley [ZS86],

which specifies systems via asynchronous processes with a finite set of states. The computations that occur during state changes in PAISley are written in a functional programming language, and so the specification of a system that uses one of the example ADTs would likely include something like the Prosper specification appearing in this paper. Although the comparison will focus on Prosper, many of the comments will also apply to these other languages, and these languages will be referred to as needed in the exposition.

Although raw Prolog is sometimes used as a specification language, Prolog is included in this comparison mostly because of the number of specification languages that make use of Prolog syntax or actual Prolog code. PLEASE [TC89], for example, uses a distinctly Prolog-like syntax, and specifications are translated to Prolog for execution in a hybrid Prolog/Ada prototyping environment. In OBSERV [TY92], finite state machines are used to model the changing state of a software system. The activities that occur on a state transition are described by Prolog code, and so a system using one of the example ADTs would likely include a section of Prolog code such as the one given in this paper. Relevant differences between PLEASE, OBSERV, and Prolog are noted in the following comparisons.

The rest of this paper is organized as follows. Sections 2.2 and 2.3 present the example specifications of ADTs Table and Bounded List, respectively, in each of the five specification languages. Some introductory material is included for readers who may not be familiar with the particular specification languages. Section 2.4 presents practical comparisons of the specification languages based on specifying the example ADTs. Section 2.5 presents more theoretical comparisons of the languages, although the example specifications are still used as illustrations. Section 2.6 sumarizes these comparisons.

2.2 Specifications of ADT Table

2.2.1 VDM

VDM uses modules for visibility control. Here, module TABLE is parameterized by RangeType, which is the type of the elements stored in the table. In VDM, *calZ* is the type name for integers. Composite types (also called trees) are defined with

the :: notation, and are basically tuples. For example, EntryType is the type of tuples with Index and Value fields. In VDM, the natural model for tables would be a map of type Index $\xrightarrow{m}$RangeType. It is modeled as a set of EntryType here to allow easier comparison with the other specifications of ADT Table, as not all of the languages have map types. Each type definition can have an invariant, provided here by the function inv-Table. The state keyword preceeds the description of the state that the operations of the module use. The initial state of Tables is included here.

The operation specifications consist of first order predicate logic pre- and post-conditions over the model types. The ext rd and ext wr clauses indicate the external variables that the operation has read and write access to, respectively. There is no requirement that these variables be global in an implementation – they can be thought of as part of the formal parameter list. Variables appearing in the ext wr clause have both a pre- and post-state. In the post-condition, the undecorated variable name denotes the post-state value and the variable name under a harpoon ($\leftharpoondown$) denotes the pre-state value. In the pre-condition, the undecorated name denotes the pre-state value.

```
module TABLE
parameters types RangeType exports
types
  Table
operations
  initTable: () →
  AddEntry: ℤ × RangeType →
  RemoveEntry: ℤ →
  AccessTable: ℤ →
  ReplaceValue: ℤ × ℤ →
definitions

types
EntryType::Index: ℤ
           Value: RangeType
Table = EntryType-set
inv inv-Table(T)
```

state
  State of T: Table
  init(mk-State(T_0)) $\triangleq$ $T_0 = \{\}$
end;
functions

inv-Table: Table $\rightarrow \mathcal{B}$
inv-Table(T) $\triangleq \forall t \in T \bullet$
  $\forall s \in T \bullet$
    $t \neq s \Rightarrow$ Index(t) $\neq$ Index(s)
operations

InitTable()
ext wr T: Table
post T = {}

AddEntry(I: $\mathcal{Z}$, V: RangeType)
ext wr AddedOk: $\mathcal{B}$, T: Table
post $((\exists e \in \overleftarrow{T} \bullet$ Index(e) = I)
    $\wedge$ T = $\overleftarrow{T}$ $\wedge$ AddedOk = false)
$\vee((\neg\exists e \in \overleftarrow{T} \bullet$ Index(e) = I)
    $\wedge$ T = $\overleftarrow{T}$ $\cup${mk-EntryType(I, V)}$\wedge$ AddedOk = true)

RemoveEntry(I: $\mathcal{Z}$)
ext wr T: Table
post T = $\overleftarrow{T}$ $-\{e \in \overleftarrow{T} |$ Index(e) = I$\}$

AccessTable(I: $\mathcal{Z}$)
ext rd T: Table
  wr V: RangeType, Defined: $\mathcal{B}$
post $((\exists e \in \overleftarrow{T} \bullet$ Index(e) = I)
    $\wedge$ V = Value(e) $\wedge$ Defined = true)
$\vee((\neg\exists e \in \overleftarrow{T} \bullet$ Index(e) = I) $\wedge$ Defined = false)

ReplaceValue(I: $\mathcal{Z}$, V: RangeType)
ext wr T: Table
  wr ChangedOk: $\mathcal{B}$

post ((∃e ∈ $\overleftarrow{T}$ • Index(e) = I
∧ T = ($\overleftarrow{T}$ −{e}) ∪ {mk-EntryType(I, V)})
∧ ChangedOk = true)
∨((¬∃e ∈ $\overleftarrow{T}$ • Index(e) = I) ∧ ChangedOk = false)

end TABLE

2.2.2 Z

In Z, schemas are used to define types and specify operations. In this example specification, schemas EntryType and Table define types and schemas AddEntry, RemoveEntry, AccessTable, ReplaceValue, and InitTable specify the operations on tables. The remaining schemas are used by these operation schemas. There are two forms for schemas: a box (labeled with the schema name) that may be separated into two pieces by a shorter horizontal line, and an equational form that uses $\hat{=}$ to define a schema name. In the box form, the part above the short horizontal line is the declaration part, and the part below the predicate part.

When a schema is used to define a type, the type's model is composed of the variables declared in the declaration, and every element of the type must satisfy the predicate part. In schema Table, the variable theTable makes up the state of Tables and the predicate is used to give the invariant on type Table. (In the type of theTable, $\mathcal{P}$ is the powerset operator.) When a schema is used to specify an operation, the declaration gives the variables used in the specification, and the predicate part defines the functionality. When the predicate of a schema specifying an operation has two parts, the first is often the pre-condition and the second the post-condition of the operation. This convention is followed in the example Z schemas.

By Z convention, undecorated variable names denote pre-state values, while primed (') variables denote post-state values. Variable names ending with a question mark denote inputs (formal parameters), while those ending with an exclaimation point denote outputs. For example, in schema KeyDefined, I? is an input and report! is an output.

The schema calculus is used to combine existing schemas in many different ways to form new schemas. Schema ΔTable, which introduces pre- and post-state identifiers

for Tables, is included in the declaration part of any schema that change the state of a Table. Similarly, schema ΞTable, which equates the pre- and post-state values, is included in the declaration of any schema which does not change the state of a Table. Including a schema in the declaration part of another in this way is equivalent to merging the declarations and conjoining the predicates of the two schemas. The other schema calculus operators used in this specification are $\wedge$ and $\vee$. The $\wedge$ operator is similar to inclusion, as S $\wedge$ T produces a schema with the declarations of S and T merged and the predicates conjoined. The result of S $\vee$ T is the schema with the declarations of S and T merged and the predicates disjoined. For example, schema AddEntry is built from schemas AddEnt, OK, and KeyDefined by using these operators.

As in VDM, Z has a map type which would have been the natural choice as the model for Table. Except for the use of the schema calculus to split up normal and error conditions, the specification is analogous to the VDM specification of ADT Table.

- EntryType
| Index: $\mathbb{Z}$
| Value: RangeType

- Table
| theTable: $\mathcal{P}$ EntryType
|
| $\forall t$:EntryType $\bullet t \in$ theTable $\Rightarrow$
| $\forall s$:EntryType $\bullet s \in$ theTable $\wedge t \neq s \Rightarrow$
| Index(t) $\neq$ Index(s)

- OK
| ΞTable
| report!: MESSAGE
|
| report! = 'Ok'

- AddEnt

| ΔTable
| I?: $\mathbb{Z}$
| V?: RangeType
| ———
| $\neg\exists e$:EntryType $\bullet e \in$ theTable(T) $\wedge$ Index(e) = I?
| $\exists e$ ::EntryType $\bullet$ Index(e) = I? $\wedge$ Value(e) = V?
| $\wedge$ theTable(T') = theTable(T) $\cup\{e\}$

- KeyDefined ———
| ΞTable
| I?: $\mathbb{Z}$
| report!: MESSAGE
| ———
| $\exists e$:EntryType $\bullet e \in$ theTable(T) $\wedge$ Index(e) = I?
| report! = 'Key already defined'

AddEntry $\hat{=}$ (AddEnt $\wedge$ OK) $\vee$ KeyDefined

- RemoveEntry ———
| ΔTable
| I?: $\mathbb{Z}$
| ———
| theTable(T') = theTable(T) $-\{e$:EntryType $\mid e \in$ theTable(T) $\wedge$ Index(e) = I?$\}$

- AccessTab ———
| ΞTable
| I?: $\mathbb{Z}$
| V!: RangeType
| ———
| $\exists e$:EntryType $\bullet e \in$ theTable(T) $\wedge$ Index(e) = I?
| $\exists e$:EntryType $\bullet e \in$ theTable(T) $\wedge$ Index(e) = I? $\wedge$ V! = Value(e)

- KeyNotDefined ———
| ΞTable
| I?: $\mathbb{Z}$
| report!: MESSAGE

| ———————
| ¬∃e :EntryType •e ∈ theTable(T) ∧ Index(e) = I?
| report! = 'Key not defined'
———————————————————

AccessTable ≙ (AccessTab ∧ OK) ∨ KeyNotDefined
- ReplaceVal ————————————————-
| ΔTable
| I?: ℤ
| V?: RangeType
| ———————
| ∃e :EntryType •e ∈ theTable(T) ∧ Index(e) = I?
| ∃e :EntryType •e ∈ theTable(T) ∧ Index(e) = I?
| ∧ theTable(T') = (theTable(T) −{e})
| ∪{e :EntryType | Index(e) = I? ∧ Value(e) = V?}
———————————————————

ReplaceValue ≙ (ReplaceVal ∧ OK) ∨ KeyNotDefined

- InitTable ————————————————
| ΔTable
| ———————
| theTable(T') = {}
———————————————————

2.2.3 SPECS-C++

SPECS-C++ specifications are intended to serve as header files for C++ implementations. Hence, the parts of the following specification that are not comments (i. e. not enclosed in the /* */ comment delimeters) are legal C++ code. In particular, the template mechanism of C++ is used to parameterize class specifications by types. The model and invariant for class Table are analogous to those for the VDM and Z versions. The type defined by a class specification is modeled as a tuple of the abstract data members (**data membersA**).

In SPECS-C++, only object types are mutable, and so only objects have pre- and post-state values. Object types end in &. For example, the type of objects containing

boolean values is bool&. An undecorated expression of an object type denotes the pre-state value of the object, while the same expression primed (') denotes the post-state value. As in C++, each member function takes an implicit argument, which in SPECS-C++ is called self, of the object or value type of the class. Thus, in the following member function specifications, self is always of type Table& or Table. The name of an abstract data member can be used as a shorthand for the corresponding field of the tuple contained in self, so theTable' is the same as self'.theTable.

Member functions with the same name as the class are constructors, and are used to initialize instances of the class. Member function Table is the only constructor of this class Table. Type void is the return type of pure procedures.

```
template<class RangeType> class Table {
/* model
**   domains
**     int DomainType
**     tuple EntryType (
**             DomainType Index,
**             RangeType Value)
**   data membersA
**     set of EntryType theTable
**   invariantA
**     \forall (EntryType t) [
**       t \in theTable =>
**         \forall (EntryType s) [
**           s \in theTable /\ t != s =>
**             Index(t) != Index(s)
**         ]
**     ]
*/
public:
Table();
/* modifies: self
** postA: theTable' = {}
*/

void AddEntry(DomainType I, RangeType V, bool& AddedOk);
/* modifies: self, AddedOk
```

```
** postA: (\exists (EntryType e) [
**           e \in theTable /\ Index(e) = I]
**         /\ AddedOk' = false)
**   \/ (!(\exists (EntryType e) [
**           e \in theTable /\ Index(e) = I])
**       /\ theTable' = theTable \union {(I, V)}
**       /\ AddedOk' = true)
*/

void RemoveEntry(DomainType I);
/* modifies: self
** postA: theTable' = theTable - {e | e \in theTable /\ Index(e) = I}
*/

void AccessTable(DomainType I, RangeType& V, bool& Defined);
/* modifies: V, Defined
** postA: (\exists (EntryType e) [
**           e \in theTable /\ Index(e) = I /\ V' = Value(e)]
**           /\ Defined' = true)
**     \/ (!(\exists (EntryType e) [
**             e \in theTable /\ Index(e) = I])
**         /\ Defined' = false)
*/

void ReplaceValue(DomainType I, RangeType V, bool& ChangedOK);
/* modifies: self, ChangedOK
** postA: (\exists (EntryType e) [
**           e \in theTable /\ Index(e) = I
**           /\ theTable' = (theTable - {e}) \union {(I, V)}]
**           /\ ChangedOk' = true)
**   \/ (!(\exists (EntryType e) [
**           e \in theTable /\ Index(e) = I])
**       /\ ChangedOk' = false)
}
```

2.2.4 Prosper

As in VDM, Prosper uses modules to control what types and operations are exported by a specification. Unlike VDM (or any other language in this survey),

types are first class citizens in the language and can be passed to and returned from functions. Many of the functions specified in this example are parameterized by type. List is a type constructor that takes a type and returns the type of lists of that type. The supertype functions SUM and SUB are used to construct new types. SUM constructs a (possibly infinite) discriminated union of types. Functions Inj and Proj are used to inject elements into and project elements from the union, respectively. SUB is used to construct subtypes by specifying a predicate that elements of the subtype must satisfy. For example, only Lists that satisfy the predicate Not_Null? typecheck as the first argument of function Head, where Not_Null? x is just not(Null? x).

As Prosper is executable, the invariant for ADT Table (as expressed by function IsTableRep?) is quite different than the invariants for the languages discussed so far, as the previous invariants have been relatively abstract. As Prosper is a pure functional language, there is no notion of pre- and post-state. The notation is LISP-like (with the exception of the type information). Prosper has only a few primitive types, so type List is specified from scratch, and then Table is built on List.

```
module Generic-List
export
  List @ TYPE → TYPE;
  Nil @ List('some:TYPE);
  Head @ SUB(Not_Null? @ (List('some:TYPE) → boolean)) → 'some;
  Tail @ SUB(Not_Null? @ (List('some:TYPE) → boolean)) → List('some);
  Cons @ ('some:TYPE × List('some)) → List('some);
  IsIn? @ 'some:TYPE → List('some) → boolean;
  Null? @ List('some:TYPE) → boolean
  Length @ List('some:TYPE) → integer
from
  value List such that List('t)
    is SUM(IsListRep? 't);
  value IsListRep? @ TYPE → TYPE → boolean such that IsListRep? 'a 'b is
    if 'b = single then true
    else if Cartesian? 'b then let 'z × 'y be 'b
      in ('z = 'a) and (IsListRep? 'a 'y)
      else false;
  value Nil is ($ @ single) @ SUM(IsListRep? ('some:TYPE));
```

```
  value Head such that Head('x) is first(Proj('x));
  value Tail such that Tail ('x:List('t))
    is (Inj (IsListRep? ('t))) (snd(Proj 'x));
  value Cons such that Cons('a, 'b:List('t))
    is (Inj (IsListRep? ('t))) ('a, (Proj 'b));
  value IsIn? such that IsIn? 'Item 'b:List('t) is
    if Null? 'b then false
    else 'Item = first(Proj('b)) or IsIn? 'Item, snd(Proj('b))
  value Null? such that Null? 'L is 'L = Nil
  value Length such that Length 'L is
    if Null? 'L then 0 @ integer
    else 1 @ integer + Length(Tail 'L)
end-from

module Table
export
  Table @ TYPE → TYPE;
  InitTable @ Table('vt:TYPE);
  AddEntry @ ('I:integer × 'vt:TYPE × SUB(Not_KeyDefined('I)
    @ Table('vt) → boolean)) → Table('vt);
  RemoveEntry @ (integer × Table('vt:TYPE)) → Table('vt);
  AccessTable @ ('I:integer × SUB(KeyDefined('I) @
    Table('vt:TYPE) → boolean)) → 'vt;
  ReplaceValue @ ('I:integer × 'vt:TYPE × SUB(KeyDefined('I)
    @ Table('vt) → boolean)) → Table('vt);
from
  value EntryType @ 'vt:TYPE → TYPE is integer × 'vt;
  value Table such that Table('vt) is SUB(IsTableRep? 'vt);
  value IsTableRep? @ 'vt:TYPE → List(EntryType('vt)) → boolean
    such that IsTableRep 'a 'T is
    if Null? 'T then true
    else if Not_IsIn? Head('T) Tail('T)
      then Not_KeyDefined first(Head('T)) Tail('T)
           and IsTableRep? 'a Tail('T)
    else IsTableRep? 'a Tail('T);
  value KeyDefined @ integer → Table('vt:TYPE) → boolean such that
    KeyDefined 'I 'T is
    if Null?('T) then false
    else first(Head('T)) = 'I or KeyDefined 'I Tail('T);
  value InitTable is Nil;
```

```
  value AddEntry such that AddEntry('I, 'V, 'T) is Cons(('I, 'V), 'T);
  value RemoveEntry such that RemoveEntry('I, 'T) is
    if Null?('T) then Nil
    else if first(Head('T)) = 'I then Tail('T)
    else Cons(Head('T), RemoveEntry('I, Tail('T)));
  value AccessTable such that AccessTable('I, 'T) is
    if first(Head('T)) = 'I then snd(Head('T))
    else AccessTable('I, Tail('T));
  value ReplaceValue such that ReplaceValue('I, 'Z, 'T) is
    if first(Head('T)) = 'I then Cons(('I, 'Z), Tail('T))
    else Cons(Head('T), ReplaceValue('I, 'Z, Tail('T)))
end-from
```

2.2.5 Prolog

A Prolog program consists of a set of rules. The part of a rule to the left of the :- is called the head, and the part to the right is the tail. The tail is optional, and a rule with no tail is called a fact. Rules define relations, so logically a fact is just an assertion that the relation holds for the case given. For example, the first rule for keyDefined asserts that the relation keyDefined holds when the first entry in the table has the given key. Rules with tails can be thought of as backwards logical implications. For example, the second rule for keyDefined can be interpreted as asserting that whenever keyDefined holds for a table, it also holds for the table constructed by adding any entry to that table. Together, these two rules define the keyDefined relation on indices and tables. As Prolog programs define relations, even the operations that are specified as functions or procedures in other languages are relations here. For example, addEntry is defined as a relation between a key and two tables, rather than an operation on tables.

In Prolog, the notation [] is used for constructing lists, and [X|Y] denotes the list with X as its first element and Y as the rest of the list. Prolog programs rely heavily on pattern matching. For example, an expression like [X|Y] can be matched with a list to decompose it. An underscore (_) is a wildcard that matches anything.

As Prolog is an untyped programming language, programs are inherently polymorphic. Although the invariant for ADT Table is not expressed here, the operations do preserve it. The invariant could be expressed and checked for any Table value.

```
keyDefined(I, [entry(I,_)|_]).
keyDefined(I, [_|TheTable]) :- keyDefined(I, TheTable).

addEntry(TheTable, I, V, [entry(I, V)|TheTable]) :-
    not(keyDefined(I, TheTable)).

removeEntry(_, [], []).
removeEntry(I, [entry(I, _)|TheTable], TheTable).
removeEntry(I, [entry(X, V)|TheTable], [entry(X, V)|NTable]) :-
    I <> X,
    removeEntry(I, TheTable, NTable).

accessTable(I, [entry(I, V)|_], V).
accessTable(I, [_|TheTable], V) :-
    keyDefined(I, TheTable),
    accessTable(I, TheTable, V).

replaceValue(I, V, [entry(I, _)|TheTable], [entry(I, V)|TheTable]).
replaceValue(I, V, [entry(X, Y)|TheTable], [entry(X, Y)|NTable]) :-
    keyDefined(I, TheTable),
    I <> X,
    replaceValue(I, V, TheTable, NTable).
```

2.3 Specifications of ADT BndList

2.3.1 VDM

This specification uses many of the same features of VDM as the example of Section 2.2.1. One difference is the inclusion of a functions section in the module. This section defines functions that are useful in constructing the operation specifications – the functions are not exported. Function definitions consist of a signature and a direct definition of the function result. Also note that the module does not give a default initial value for BndLists. Instead, the operation InitBndList is provided for initializing them. This approach allows the user to set the maximum size of the BndList when it is initialized.

module BNDLIST

parameters types ItemType
exports
types
  BndList
operations
  InitBndList: $\mathcal{N}_1 \rightarrow$
  InitCursor: () $\rightarrow$
  AdvanceCursor: () $\rightarrow$
  CursorOffEnd: () $\rightarrow \mathcal{B}$
  CurrentItem: () $\rightarrow$ ItemType
  InsertFront: ItemType $\rightarrow$
  RemoveAllOccurrences: ItemType $\rightarrow$
definitions
types
BndList::theList: ItemType*
        currCursor: $\mathcal{N}_1$
        maxSize: $\mathcal{N}_1$
inv(mk-BndList(theList, currCursor, maxSize)) $\triangleq$
        currCursor $\leq$ len theList + 1 $\wedge$ len theList $\leq$ maxSize

state
  State of L: Bndlist
end;

functions

OffEnd: BndList $\rightarrow \mathcal{B}$
OffEnd(L) $\triangleq$ (currCursor(L) = len theList(L) + 1)

WithItemRemoved: ItemType* $\times$ ItemType $\rightarrow$ ItemType*
WithItemRemoved(LP, Item) $\triangleq$
if LP = []
  then []
  else if hd(LP) = Item
    then WithItemRemoved(tl(LP), Item)
    else [hd(LP)] $\frown$ WithItemRemoved(tl(LP), Item)

operations

InitBndList(initialMaxSize:$\mathcal{N}_1$)
ext wr L: BndList
post theList(L) = [] $\wedge$ currCursor(L) = 1 $\wedge$ maxSize(L) = initialMaxSize

InitCursor()
ext wr L: BndList
post currCursor(L) = 1 $\wedge$ theList(L) = theList($\overleftarrow{\text{L}}$)
$\wedge$ maxSize(L) = maxSize($\overleftarrow{\text{L}}$)

AdvanceCursor()
ext wr L: BndList
pre $\neg$ OffEnd(L)
post currCursor(L) = currCursor($\overleftarrow{\text{L}}$) + 1 $\wedge$ theList(L) = theList($\overleftarrow{\text{L}}$)
$\wedge$ maxSize(L) = maxSize($\overleftarrow{\text{L}}$)

CursorOffEnd() b: $\mathcal{B}$
ext rd L: BndList
post b = OffEnd(L)

CurrentItem() Item: ItemType
ext rd L: BndList
pre $\neg$ OffEnd(L)
post Item = theList(L)(currCursor(L))

InsertFront(Item: ItemType)
ext wr L: BndList
pre len theList(L) < maxSize(L)
post theList(L) = [Item] $\frown$ theList($\overleftarrow{\text{L}}$) $\wedge$ currCursor(L) = currCursor($\overleftarrow{\text{L}}$) + 1
$\wedge$ maxSize(L) = maxSize($\overleftarrow{\text{L}}$)

RemoveAllOccurrences(Item: ItemType)
ext wr L: BndList
post theList(L) = WithItemRemoved(theList($\overleftarrow{\text{L}}$), Item) $\wedge$ currCursor(L) = 1
$\wedge$ maxSize(L) = maxSize($\overleftarrow{\text{L}}$)

end BNDLIST

2.3.2 Z

The Z features demonstrated in this example specification not appearing in the example of Section 2.2.2 are: including schemas in the predicate part of another schema and the sequence operators $\lceil$ and rng. The inclusion of schema OffEnd in the predicate of schema AdvanceCursor is equivalent to substituting the predicate of OffEnd for the name. In this case the schemas used the same variable names, but the schema calculus provides a renaming operator for cases where different variable names are used. Operator $\lceil$ (used in schema RemoveAllOccurrences) is used to restrict the range of a sequence (i. e. the items contained in the sequence) to a specified set. The result sequence is "squashed" so that no gaps appear in it. The following example illustrates this.

$$[1,2,3,4,2,1]\lceil\{1,3\} = [1,3,1]$$

The rng operator returns the range of its argument sequence as a set, as shown in schema RemoveAllOccurrences.

- BndList ———————————-
| theList: seq ItemType
| currCursor: $\mathcal{N}^+$
| maxSize: $\mathcal{N}^+$
| ———
| currCursor $\leq$ #theList + 1 $\wedge$ #theList $\leq$ maxSize
———————————————

- OffEnd ———————————
| L: BndList
| ———
| currCursor(L) = #theList(L) + 1
———————————————

- InitBndList ———————————
| ΔBndList
| initialMaxSize?: $\mathcal{N}^+$
| ———
| initialMaxSize? > 0
| theList(L') = []$\wedge$ currCursor(L') = 1$\wedge$ maxSize(L') = initialMaxSize?

- InitCursor ———————————
| ΔBndList
| ———————
| currCursor(L') = 1 ∧ theList(L') = theList(L) ∧ maxSize(L') = maxSize(L)

- AdvanceCursor ———————————
| ΔBndList
| ———————
| ¬ OffEnd
| currCursor(L') = currCursor(L) + 1 ∧ theList(L') = theList(L)
∧ maxSize(L') = maxSize(L)

- CursorOffEnd ———————————
| ΞBndList
| Out!: boolean
| ———————
| Out! = OffEnd

- CurrentItem ———————————
| ΞBndList
| Out!: ItemType
| ———————
| ¬ OffEnd
| Out! = theList(L)(currCursor(L))

- InsertFront ———————————
| ΔBndList
| Item?: ItemType
| ———————
| #theList(L) < maxSize
| theList(L') = [Item?] ⌢ theList(L) ∧ currCursor(L') = currCursor(L) + 1
| ∧ maxSize(L') = maxSize(L)

- RemoveAllOccurrences ————————————
| ΔBndList
| Item?: ItemType
| ————————
| theList(L') = theList(L) ⌈ (rng(theList(L)) - {Item?})
| ∧ currCursor(L') = 1 ∧ maxSize(L') = maxSize(L)
————————————————————————————

2.3.3 SPECS-C++

The SPECS-C++ features demonstrated in this specification that do not appear in the specification of Section 2.2.3 include the definition and use of abstract functions and the specification of member functions with non-void return types. Abstract functions are not intended to be implemented. Rather, they are used in the specification of the member functions in much the same way that functions are used by the operation specifications in the VDM specification of ADT Bounded List (Section 2.3.1). Specifications of member functions with non-void return types use the SPECS-C++ keyword result to refer to the return value. Otherwise, such specifications are the same as any other member function specification.

```
template<class ItemType> class BndList {
/* model
**   domains
**     type ListPart sequence of ItemType
**   data membersA
**     ListPart theList
**     int currCursor
**     int maxSize
**   invariantA
**     1 <= currCursor /\ currCursor <= length(theList) + 1
**     /\ length(theList) <= maxSize /\ maxSize > 0
**   abstract functions
**     define OffEnd(BndList L) as bool s.t.
**       OffEnd(L) = (L.currCursor = length(L.theList) + 1)
**     define WithItemRemoved(ListPart LP, ItemType Item) as ListPart
**   s.t.  (LP = <> => WithItemRemoved(LP, Item) = <>)
```

```
**       /\ (LP != <> =>
**            (first(LP) = Item => WithItemRemoved(LP, Item) =
**                                 WithItemRemoved(trailer(LP), Item))
**         /\ (first(LP) != Item => WithItemRemoved(LP, Item) =
**                          <first(LP)> || WithItemRemoved(trailer(LP),
**                                                         Item)))
*/
public:
BndList(int initialMaxSize);
/* preA: initialMaxSize > 0
** modifies: self
** postA: theList' = <> /\ currCursor' = 1 /\ maxSize' = initialMaxSize
*/

void InitCursor();
/* modifies: self
** postA: currCursor' = 1
*/

void AdvanceCursor();
/* preA: !OffEnd(self)
** modifies: self
** postA: currCursor' = currCursor + 1
*/

bool CursorOffEnd();
/* PostA: result = OffEnd(self)
*/

ItemType CurrentItem();
/* preA: !OffEnd(self)
** postA: result = index(theList, currCursor)
*/

void InsertFront(ItemType Item);
/* preA: length(theList) < maxSize
** modifies: self
** postA: theList' = <Item> || theList /\ currCursor' = currCursor + 1
*/
```

```
void RemoveAllOccurrences(ItemType Item);
/* modifies: self
** postA: theList' = WithItemRemoved(theList, Item) /\ currCursor' = 1
*/
}
```

2.3.4 Prosper

The Prosper specification of ADT Bounded List uses the List type exported by module Generic-List (Section 2.3.4). The main language feature not seen in that previous Prosper specification is the use of an explicit cartesian product type for the representation of type BndList.

```
module BList
export
  BndList @ TYPE → TYPE;
  InitBndList @ integer → BndList('it:TYPE);
  InitCursor @ BndList('it:TYPE) → BndList('it);
  AdvanceCursor @ SUB(Not_CursorOffEnd @ BndList('it:TYPE) → boolean)
    → BndList('it);
  CursorOffEnd @ BndList('it:TYPE) → boolean;
  CurrentItem @ SUB(Not_CursorOffEnd @ BndList('it:TYPE) → boolean) → 'it;
  BLength @ BndList('it:TYPE) → integer;
  InsertFront @ (('it:TYPE) × SUB(ShortList? @ BndList('it) → boolean))
    → BndList('it);
  RemoveAllOccurrences @ ('it:TYPE × BndList('it)) → BndList('it);
from
  value BndList such that BndList('it) is
    SUB(isBndListRep? 'it);
  value isBndListRep? @ 'it:TYPE → (List('it) × integer × integer) → boolean
    such that isBndListRep? 't ('B × 'size × 'maxsize) is
    'size > 0 @ integer and 'size <= Length('B) + 1 @ integer
    and 'maxsize > 0 @ integer and Length('B) <= 'maxsize;
  value InitBndList such that InitBndList('initialMaxSize) is
    (Nil, 1 @ integer, 'initialMaxSize);
  value InitCursor such that InitCursor('B) is
    (first('B), 1 @ integer, third('B));
  value AdvanceCursor such that AdvanceCursor('B) is
```

```
    (first('B), snd('B) + 1 @ integer, third('B));
  value CursorOffEnd such that CursorOffEnd('B) is
    snd('B) = Length(first('B) + 1 @ integer);
  value CurrentItem such that CurrentItem('B) is
    if snd('B) = 1 then Head(first('B))
    else CurrentItem(Tail(first('B)), snd('B) - 1 @ integer, third('B));
  value BLength such that BLength('B) is Length(first('B));
  value ShortList? @ BndList('it:TYPE) → boolean such that ShortList? 'B is
    BLength('B) < third('B);
  value InsertFront such that InsertFront('Item, 'B) is
    (Cons('Item, first('B)), snd('B) + 1 @ integer, third('B));
  value WithItemRemoved @ (List('it:TYPE) × 'it) → List('it)
    such that WithItemRemoved('LP, 'Item) is
    if 'LP = Nil then Nil
    else if Head('LP) = 'Item
      then WithItemRemoved(Tail('LP), Item)
      else Cons(Head('LP), WithItemRemoved(Tail('LP), 'Item));
  value RemoveAllOccurrences such that RemoveAllOccurrences('Item, 'B) is
    (WithItemRemoved(first('B), 'Item), 1 @ integer, third('B))
end-from
```

2.3.5 Prolog

The Prolog version of ADT Bounded List uses largely the same language features as those seen in Section 2.2.5. Here, terms of the form bndlist(L, CurrCursor, MaxSize) are used to represent bounded lists. In Prolog terminology, such a term is a tree, and bndlist is an (uninterpreted) functor.

```
length([], 0).
length([_|L], N) :-
    length(L, N1),
    N is N1 + 1.

offEnd(bndlist(TheList, CurrCursor, _)) :-
    length(TheList, Len),
    CurrCursor is Len + 1.

withItemRemoved([], _, []).
```

```
withItemRemoved([Item|LP], Item, NLP) :-
    withItemRemoved(LP, Item, NLP).
withItemRemoved([X|LP], Item, [X|NLP]) :-
    X <> Item,
    withItemRemoved(LP, Item, NLP).

isInList(bndlist([Item|_], _, _), Item).
isInList(bndlist([_|LP], CurrCursor, MaxSize), Item) :-
    isInList(bndlist(LP, CurrCursor, MaxSize), Item).

initCursor(bndlist(TheList, _, MaxSize), bndlist(TheList, 1, MaxSize)).

advanceCursor(bndlist(TheList, CurrCursor, MaxSize),
              bndlist(TheList, NC, MaxSize)) :-
    not(offEnd(bndlist(TheList, CurrCursor, MaxSize))),
    NC is CurrCursor + 1.

cursorOffEnd(Self) :- offEnd(Self).

currentItem(bndlist([Item|_], 1, _), Item).
currentItem(bndlist([X|LP], CurrCursor, MaxSize), Item) :-
    not(offEnd(bndlist([X|LP], CurrCursor, MaxSize))),
    NC is CurrCursor - 1,
    currentItem(bndlist(LP, NC, MaxSize), Item).

insertFront(bndlist(TheList, CurrCursor, MaxSize), Item,
            bndlist([Item|TheList], NC, MaxSize)) :-
    length(TheList, Len),
    Len < MaxSize,
    NC is CurrCursor + 1.

removeAllOccurrences(bndlist(TheList, _, MaxSize), Item,
                     bndlist(LP, 1, MaxSize)) :-
    withItemRemoved(TheList, Item, LP).
```

2.4 Practical Comparisons

This section compares the various specification languages in practical terms, examining issues that would arise in day-to-day use of the languages. The specific issues

included are: handling invalid inputs, support for generic specifications, handling of side effects, framing, support for re-use of specifications, and ease of use.

2.4.1 Handling Invalid Inputs

In this section we discuss support for specifying what happens when an operation is invoked on invalid or unexpected data. For example, consider calling AddEntry (Section 2.2) with a key that is already defined in the table, or AdvanceCursor (Section 2.3) when the cursor is at the end of the list. In VDM, the specifier basically has two choices: use a strong pre-condition that such input will not satisfy, in which case the result is unspecified, or weaken the pre-condition and handle exceptional conditions in the post-condition, perhaps by setting some error flag. This two alternatives are demonstrated in the example VDM specifications: in module BNDLIST (Section 2.3.1), strong pre-conditions are used so that no error handling is needed in the post-conditions, while in module TABLE (Section 2.2.1), all pre-conditions are just true, and exceptional conditions are handled in the post-condition by setting error flags. In SPECS-C++, the same two alternatives are available to the specifier, and the example specifications are analogous to the VDM version in this respect (Sections 2.3.3 and 2.2.3).[1]

In Z, the specifier again has the same options, and the use of explicit pre-conditions is demonstrated in the specification of ADT Bounded List (Section 2.3.2). However, the specifier can also use the schema calculus to separate normal operation from handling invalid inputs (in this case, setting an error flag as in the other specifications), and this alternative is demonstrated in the specification of ADT Table (Section 2.2.2). In the specification of AddEntry, the schema AddEnt gives the pre-condition for normal operation, and specifies the resulting table. The pre-condition of schema KeyDefined is the compliment of AddEnts, and KeyDefined specifies what happens under exceptional circumstances – in this case, that an error is reported. AddEntry is composed from these two schemas, and an additional schema OK that specifies reporting of normal behavior. This is an elegant way of composing specifi-

[1] Future plans for SPECS-C++ include adding the ability to specify C++ exceptions as in Larch/C++ [LC93], which will give the specifier another option.

cations, and gives some hint of the power of the schema calculus.

In Prosper, no explicit pre-conditions are used. Instead, the dependent type system is used to ensure that if the arguments to the function typecheck, then the function can handle them. For example, in the specification of AdvanceCursor in module BList (Section 2.3.4), the type of the BndList argument is constrained to be only those lists that satisfy the predicate Not_CursorOffEnd. This corresponds to the pre-condition of the VDM specification of AdvanceCursor.

Prolog has no explicit pre-conditions, but predicates can easily be written with "guards" so that the predicates will fail if the corresponding pre-condition is not satisfied. For example, the predicate for accessTable in the Prolog Table specification (Section 2.2.5) uses predicate keyDefined, which is equivalent to the corresponding SPECS-C++ pre-condition in class Table (Section 2.2.3). In PLEASE, separate pre- and post-condition predicates are defined, but for execution are translated into a single Prolog rule with the pre-condition serving as a guard. As OBSERV specifications use raw Prolog code, specifiers would be free to use this pre- and post-condition style. However, it does not seem likely that specifiers would make the mental distinction between pre- and post-condition predicates. The mindset underlying OBSERV seems to be that these parts of specifications are simply Prolog code, and not representations of pre- and post-condition style specifications.

2.4.2 Support for Generic Specifications

Generic specifications refer to specifications that are parameterized by types, or that allow polymorphism in the specification. For example, is it possible to write a Bounded List specification where the type of the list elements is not known? For all of the languages included here, the answer is yes. However, the techniques used in each language differ, and so are worthy of some discussion.

VDM modules (discussed in Section 2.5.1) can be parameterized by types, and both example specifications (Sections 2.2.1 and 2.3.1) demonstrate this feature. The example Z specifications (Sections 2.2.2 and 2.3.2) simply introduce a new type name and use it consistently in specifying each data type. SPECS-C++ template classes (Sections 2.2.3 and 2.3.3) are similar to VDM modules, as they allow the ADT specifications to be explicitly parameterized by types. Prosper has an even more general

notion of type parameters, as types are first class citizens in the language – they can be passed to and returned from functions. All of the example Prosper modules (Sections 2.2.4 and 2.3.4) are parameterized by types. Finally, Prolog (Sections 2.2.5 and 2.3.5) is an untyped programming language, and so is inherently polymorphic.

A crucial deficiency of all of these languages is the lack of any support for specifying characteristics that any actual type substituted for the parameter type must possess. For example, consider specifying ADT Table with the domain type as a parameter (rather than just integer). Clearly, any actual type substituted for this parameter type must allow comparison via equality – otherwise, there is no way to retrieve something from a table by its key. However, none of the languages compared here provide the ability to specify that a parameter type allows such comparisons. Some existing specification languages, for example Larch/C++ and the language described in [EHMO91], do allow specifying such characteristics of parameter types, and research aimed at adding this feature to SPECS-C++ is underway.

2.4.3 Handling of Side Effects

As most programs written in imperative languages rely on side effects for efficiency, it is appropriate to look at how well the various languages allow specification of side effects. VDM, Z, and SPECS-C++ have a built-in notion of state, and all provide similar mechanisms for referring to the pre- and post-state values of variables, as illustrated in the example specifications. Hence, they are well suited for specifying programs with side effects. The example specifications make only trivial use of SPECS-C++'s object types, and so do not demonstrate the power that they give to the specifier. For example, in SPECS-C++ the BndList template could be instantiated with an object type: `BndList<int&>` is the type of bounded lists of objects containing integers. Any external mutation of an integer object contained in such a bounded list would be visible through the list. Prosper (and any pure functional programming language) and Prolog have no explicit notion of state. If the specifier wishes to specify state changes, all the operations that change the state must take a "state" argument, and return a "state" result. Both example specifications in both languages take this approach, thus cluttering up the specification. Even in PLEASE, where there is a notion of pre- and post-state, the predicates that the specifier defines

typically take this form. Hence, these languages are not as well suited as the other three for specifying operations with side effects.

2.4.4 Framing

The framing problem in operation specifications is the difficulty of saying "and nothing else changes" [BMR93]; that is, the problem of finding compact and elegant techniques for specifying what parts of the state can change and what parts can't.

VDM has ext rd and ext wr clauses, which specify which external variables the specified operation is allowed to read and write, respectively. Thus, the value of a variable can only change from the pre- to the post-state if it is listed in the ext wr clause. The example specifications (Sections 2.2.1 and 2.3.1) demonstrate the use of these clauses.

By Z convention, schemas with shape Δ are used to define pre- and post-state (primed) variable names for a particular type, and schemas with shape Ξ are used to specify that these variables are equal. That is, schemas with shape Δ are used to introduce pre- and post-state declarations for variables that can change. This schema is then included in any schema that can change the state of these variables. For example, ΔBndList (Section 2.3.2) is such a schema, and the schema InitBndList includes this schema. The appropriate schema of shape Ξ can be included in an operation schema to specify that the included variables do not change value from the pre- to the post-state. Both Z specifications use such a schema. For example, the Z specification of ADT Table (Section 2.2.2) defines such a schema, and includes it in schemas KeyDefined and KeyNotDefined to specify that the state of the table does not change under these exceptional circumstances. This makes explicit which schemas can modify the state of elements of what types.

SPECS-C++ provides two aids to framing. The first is the modifies clause, which explicitly states what objects the specified operation is allowed to change. Thus, it is similar to the kind of support for framing provided in VDM and Z. Many of the example SPECS-C++ operation specifications (Sections 2.2.3 and 2.3.3) demonstrate the use of the modifies clause. The second SPECS-C++ aid to framing is the default "nothing else changes" semantics, which says that an object does not change its value from pre-state to post-state, unless the specification forces it to. This semantics is

used in the specification of operation AddEntry in class Table, where the post-state value of the table is not defined when the post-state value of AddedOk is false. The "nothing else changes" semantics allows the reader to infer that the state of the table did not change. Note that VDM and Z have no analog to this semantics, so if a specification does not specify a post-state value for a variable that it has "write access" to, an implementation is free to give an arbitrary post-state value for it. Hence, all the example VDM and Z specifications provide explicit post-state values for such variables by equating the pre- and post-state values.

Both Prosper and Prolog are side effect free, and so have no notion of pre- and post-state. However, as previously noted, many of the operations in the example specifications take a state argument, and return a state result, so that state changes are modeled by returning a new state that corresponds to the post-state of the SPECS-C++ specification. Under this modeling of state, Prosper and Prolog have no support for framing. As the notion of state is not built into the language, there is no reason to expect any correspondence between a state argument and a state result other than that explicitly specified. Thus, the specifier must completely specify a state result, even when that result is the same as a state argument.

2.4.5 Support for Re-use of Specifications

As formal specification is a time consuming and expensive process, it is important that specifications and parts of specifications can easily be re-used. All of the languages studied here allow some form of re-use, but the mechanisms used vary widely. In VDM, pre- and post-conditions and invariants can be used as functions. For example, the pre-condition of operation AdvanceCursor used as a function would have the name pre-AdvanceCursor. Additionally, the types and operations that a module exports can be used in other specifications. In Z, the schema calculus provides many powerful operators for combining existing schemas to produce new ones. The Z specification of ADT Table illustrates the use of the $\wedge$ and $\vee$ operators for combining schemas (Section 2.2.2). Additionally, operators are provided for extracting parts of schemas so that they can be re-used.

SPECS-C++ supplies two mechanisms for specification re-use - specification inheritance and the ability to use member function return values and side effects in

specifications. Specification inheritance is unique to SPECS-C++ among the languages studied here, but is similar to the specification inheritance found in some other object-oriented specification languages such as Fresco.

Since Prosper is based on functional programming languages, and Prolog is a programming language, both allow programming language kinds of re-use – functions and predicates that have been defined can be called as often as needed. In Prosper, the types and functions that a module exports can be used in the specification of another module. For example, the module Generic-List (a slightly modified version of the one appearing in [LB88]) is used both in the specification of module Table (Section 2.2.4) and module BList (Section 2.3.4). In both Prolog specifications, some of the predicates are used several times in the specification of other predicates.

As OBSERV is an object-oriented (more properly, object-based, as inheritance is not supported), it defines objects that can be re-used. Additionally, objects can define types, and these types can be used and re-used as needed. PLEASE packages (which are similar to modules) and the types they define can be re-used in other PLEASE specifications. PAISley apparently provides no support for re-use.

2.4.6 Ease of Use

This section discusses the human interface of the various languages. A number of issues are pertinent: how difficult the language is to learn, how difficult it is to write clear and concise specifications in the language, and so on.

VDM, Z, and SPECS-C++ all use similar mathematical models (set, sequence, tuple, ...) and similar specification techniques (pre- and post-conditions written in first order predicate logic), and so are quite similar in usability. SPECS-C++ specifications that make sophisticated use of objects can become involved, but as the example specifications show, objects do not complicate specifications that do not need to make explicit use of objects. The schema calculus and various schema shapes add a level of complexity to Z specifications. However, Z does not force any sophisticated use of these features in writing reasonable specifications, so this complexity need not be a burden to novice Z specifiers. For experienced specifiers, use of the schema calculus makes writing specifications less tedious, as existing schemas can easily combined to produce new schemas.

For those with some experience in functional programming, Prosper is reasonably easy to use. However, some of the notations used for handling dependent types seem non-intuitive and difficult to learn. The amount of algorithmic information required can also become a burden on the specifier. For example, the definition of IsTableRep? in module Table (Section 2.2.4), requires a quite subtle transformation of the invariant for the corresponding ADT Table specification in VDM (Section 2.2.1), as an assertion involving two universal quantifiers must be transformed into a recursive function.

One initial difficulty in using Prolog is that Prolog programs use relations, rather than functions. For example, many people would most naturally specify or implement removeAllOccurrences (Section 2.3.5) as a function, rather than a relation between bndLists. Note that the other languages specify RemoveAllOccurrences as a function. However, a bit of experience with Prolog is usually all that is required to overcome this problem.

A much greater problem in using Prolog is understanding Prolog's non-logical features, such as not, cut, fail, and so on. Both example Prolog specifications use the not predicate, which is defined in Prolog as not(X) ≡ X, !, fail. In other words, not(X) succeeds when X fails, which is different than succeeding when X is false. For example, the query currentItem(B, 3) (see Section 2.3.5) asking for bndlists whose cursor points at 3, will either cause an error or return at most one answer depending on the Prolog implementation being used. This occurs because an unbound variable is used inside the argument to not. Clearly, this can not be explained by the logical interpretation, and so the programmer is forced to fall back on the procedural interpretation. In our experience as learners and teachers of Prolog, the procedural interpretation is much more difficult for humans to understand than the logical interpretation. Unfortunately, non-trivial use of Prolog usually requires understanding the procedural interpretation.

PLEASE avoids the use of the procedural interpretation by excluding the specification of negative information (the usual cause of falling back on the procedural interpretation) from PLEASE specifications. For example, the specifier is not allowed to use not in specifications, and would have to find another way to specify currentItem – perhaps by writing an explicit notOffEnd predicate. Although this does avoid the procedural interpretation, finding clever ways to avoid negation is clearly a burden

on the specifier.

2.5 Theoretical Comparisons

This section compares the specification languages in terms of differences arising from the design philosophies behind and the intended usage of the languages. These differences include support for data abstraction, handling of invariants, level of abstraction, and executability.

2.5.1 Support for Data Abstraction in Specifications

They key to data abstraction is the ability to hide the representation of data types. In the context of formal specifications, this corresponds to hiding the model of an ADT from parts of the specification outside of the ADT. Data abstraction in the specification of ADTs is useful for the same reasons that it is useful in the implementation of ADTs, as it allows the representation of the data type to be changed in the specification of the ADT without affecting parts of the specification outside of the ADT specification. Such a change would be useful, for example, if the specification of additional operations of the ADT would be eased by changing the representation. All of the languages in this study except Z, SPECS-C++, raw Prolog, and PAISley provide some way to use data abstraction in specifications.

VDM specifies data abstraction through modules, which bind together a model and a collection of operations. Import and export lists control what operations and types are visible to the outside world, and what outside operations are used in a module. The model is hidden inside the module. The example VDM specifications are both written as modules (Sections 2.2.1 and 2.3.1).

In Z, when a schema is used to compose a type, the variables that make up the type and the types of those variables are not hidden from other schemas. For example, any schema that declares a variable of type Table (Section 2.2.2) has access to the to the representation of tables. The schema calculus operator $\setminus$ can be used to hide a specified set of the variables the variables of a schema, but there does not seem to be any reasonable way to use this feature to enforce data abstraction in the specification.

As SPECS-C++ is an interface specification language, the data abstraction present in implementations can be specified by the class and member protection mechanisms of C++. For example, both **public** (globally visible) and **protected** member functions (visible only to derived classes) of a class can be specified. The development methodology associated with SPECS-C++ goes beyond these mechanisms, as it requires that all data members in the implementation of a specification be private, and so inaccessible to clients and derived classes of the implementation. On the other hand, the abstract data members used in specifications are visible throughout the specification. Thus, SPECS-C++ does not support data abstraction in specifications.

In Prosper, data abstraction is provided by modules with explicit export lists of operators and types visible outside the module. Both of the example Prosper specifications (Sections 2.2.4 and 2.3.4) use this feature. PAISley provides no explicit way to specify data abstraction. This limitation is noted by Zave and Schell [ZS86], and identified as a weakness of the language.

Raw Prolog provides no data abstraction, but both of the Prolog-based languages in this study do. In PLEASE, data abstraction can be specified via packages, which are modeled after Ada packages. As Ada packages are similar to modules as discussed above, the data abstraction provided is similar. Since PLEASE was designed for prototyping Ada implementations, this data abstraction is apparently present in both the specification and implementation. In OBSERV, compound objects composed of collections of other objects can be defined. Such compound objects have a well-defined interface. Additionally, the names of the component objects are not visible in the parts of the specification outside the compound object. Thus, data abstraction is present in the specification.

2.5.2 Invariants

VDM, Z, SPECS-C++, and Prosper each provide the notion of a data invariant as a predicate that members of the specified ADT must satisfy in any state that is ovservable from outside of the ADT. Prolog does not, so instead the discussion focusses on how PLEASE and OBSERV specify invariants.

In VDM an invariant can be defined for any type that is declared. For example,

the invariant for type Table (Section 2.2.1) says that the same key is not present in two distinct elements of a table. The invariant for type BndList (Section 2.3.1) describes legal relationships among the sub-components that make up the composite value. In Z, when a schema defines a type, the predicate part (the part below the short horizontal line) can be used to define an invariant on that type. This is done in both schemas Table and BndList (Sections 2.2.2 and 2.3.2). Similarly, SPECS-C++ invariants describe the legal states of the abstract data members of a class. Thus, the SPECS-C++ invariants (Sections 2.2.3 and 2.3.3) are similar to the VDM and Z invariants, except that the lack of a type like $\mathcal{N}^+$ makes the invariant for class BndList slightly longer.

In Prosper, invariants are an integral part of the type being specified. For BList, the type operator SUB is used to express the invariant in the definition of type BndList (Section 2.3.4). This is even clearer in modules Generic-List and Table (Section 2.2.4), where the functions IsListRep? and IsTableRep? that express the appropriate invariants are explicitly used as arguments to SUB in the type definitions. This use of the type system for preserving invariants is one of the greatest strengths of Prosper.

PAISley provides no support for specifying invariants. Support for automated consistency checking of declarations, definitions, and applications is provided, but this largely corresponds to the typechecking done by most interpreters and compilers.

In PLEASE, packages that define data types can also provide an explicit invariant for that type. However, it is unclear whether these invariants are ever executed. OBSERV currently provides no support for invariants, but they are discussed as a possible improvement to the language.

2.5.3 Level of Abstraction

The non-executable specification languages (VDM, Z, and SPECS-C++) all use pre- and post-condition style specification (although VDM also permits a more explicit style), and use a similar vocabulary in terms of the underlying model. As the pre- and post-conditions are expressed in the first order predicate calculus, such specifications can be quite abstract. The schema calculus of Z provides an additional level of abstraction, as schemas (specifications) can be viewed as building blocks for other specifications.

As the execution of Prosper (and many of the executable subsets of VDM) is based on functional programming, it is much more abstract than implementations in a typical imperative programming language would be, and the example specifications show this. However, Prosper (and again the other languages mentioned) does require the specifier to provide algorithms, and so is considerably less abstract than VDM, Z, or SPECS-C++. The example specifications are too short to show this clearly, but the function IsTableRep? (Section 2.2.4) does point the way, as it is considerably less abstract than the corresponding VDM invariant for module Table. This algorithmic information is a burden on the specifier and a potential source of implementation bias.

While one could argue that, as Prolog is based on first order predicate logic, the Prolog based languages (PLEASE and OBSERV) are just as abstract as VDM, Z, or SPECS-C++, in practice this is not true. Prolog specifications require the specifier to give enough information for the Prolog interpreter to function, and, as shown in the specification of ADT Table (Section 2.2.5), the level of information required frequently goes beyond the logical interpretation of Prolog. The goal is laudable, but the implementation falls short.

2.5.4 Executability

Of the languages in this study, only VDM, Z, and SPECS-C++ are not directly executable. Work on executing VDM has concentrated on defining executable subsets. In general, either the executable subset is an (algorithmic) subset of explicit specifications [Hen86, RE93, LL91, AELL92], or else implicit specifications are constrained to be written in an "algorithmic enough" style for execution [HI88, HI86, O'N92a, O'N92b]. Work on executing Z seems to have concentrated on translating specifications to Prolog [DKC90, WE92] or to some functional programming language [JS90]. These techiques are limited to the subset of Z that is amenable to fairly direct translation.

The executable subset of SPECS-C++ [WBL94] is considerably larger than that of any of the subsets of VDM and Z cited, as it includes implicit specifications that use many forms of quantified assertions. Even many quantified assertions with references to post-state values in their bodies can be executed. The SPECS-C++ specifications

included in this study (Sections 2.2.3 and 2.3.3) are both directly executable by the SPECS-C++ execution technique. Thus, executable SPECS-C++ specifications can be written at a much higher level of abstraction than that forced by specification languages such as Prosper that rely on functional programming for executability. The SPECS-C++ execution technique also avoids the problems encountered in Prolog specifications, such as the use of Prolog syntax and the non-logical features of Prolog. Research into using backtracking and constraint satisfaction techniques in executing SPECS-C++ has given the executable subset of SPECS-C++ a level of expressiveness very similar to that of Prolog.

2.6 Conclusion

Table 2.1 summarizes the comparison results for the criteria that showed significant differences. Of particular note is the power of Z's schema calculus, as it makes Z best in terms of specification re-use. Even SPECS-C++'s specification inheritance can not completely match Z's ability to use and extend existing specifications. SPECS-C++ is strongest in language support for framing and specifying side effects. Prosper is best in terms of the handling of invariants, as the invariant is an integral part of the ADT defined, and is automatically checked each time an element of the data type is created.

The main contribution of this case study is to highlight some of the relative strengths and weaknesses of the included languages (and the approaches to formal specification that they represent) in specifying ADTs. While the criteria used in this comparison are certainly not exhaustive, they do allow a reasonable comparison of the features of these languages. Also included is a brief introduction to the five main languages and examples of typical specifications in these languages. This study should be helpful in choosing a specification language for a particular specification situation, in designing new specification languages, and in teaching and using these specification languages.

Table 2.1: Comparisons of languages on selected criteria (NA = not applicable)

Criteria	VDM	Z	SPECS-C++	Prosper	PAISley
Support for Specifying Side Effects	yes	yes	yes	no	no
Support for Framing					
– limited access to state	yes	yes	yes	NA	NA
– semantic	no	no	yes	NA	NA
Support for Re-use	medium	high	high	medium	low
Ease of Use	high	high	high	medium	medium
Support for Specifying Data Abstraction	yes	no	no	yes	no
Level of Abstraction	high	high	high	medium	medium
Executability	low	low	medium	high	high

Criteria	Prolog	Please	OBSERV
Support for Specifying Side Effects	no	no	no
Support for Framing			
– limited access to state	NA	no	NA
– semantic	NA	no	NA
Support for Re-use	low	medium	medium
Ease of Use	medium	medium	medium
Support for Specifying Data Abstraction	no	yes	yes
Level of Abstraction	medium	medium	medium
Executability	high	high	high

3. EXECUTING FORMALIZED DATA FLOW DIAGRAM SPECIFICATIONS

A paper in preparation for the Journal of Information Systems

Tim Wahls[1], Albert L. Baker, and Gary T. Leavens

Abstract

While traditional Data Flow Diagrams (DFDs) are popular, they do not specify software systems precisely or unambiguously. DFD-SPECS is a formalization of DFDs for specifying concurrent software. DFD-SPECS relies on model-based specification of abstract data types, and processes are specified with first order assertions. We provide an interpreter for a subset of DFD-SPECS that includes many quantified assertions. The interpreter helps formalize the semantics of DFD-SPECS, makes possible the validation of specifications, and allows a specification to serve as a prototype for its implementation.

keywords: Data Flow Diagrams, rapid prototyping, validation of specifications, formal specifications, executable specifications, concurrent systems, formal semantics

---

[1]Wahls's work is supported in part by a research assistantship provided by the College of Liberal Arts and Sciences, Iowa State University, and in part by a fellowship provided by IBM Rochester. Baker's work was supported in part by a grant from Rockwell International. Leavens's work is supported in part by the National Science Foundation under Grant CCR-9108654.

Authors' address: Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Ames, Iowa 50011-1040 USA
telephone: (816) 562-3270
email: wahls@cs.iastate.edu baker@cs.iastate.edu leavens@cs.iastate.edu

3.1 Introduction

3.1.1 Data Flow Diagrams

Traditional Data Flow Diagrams (DFDs) are probably the most widely used specification technique in industry today. They are the cornerstone of the software development methodology commonly referred to as "Structured Analysis" (SA). Their popularity arises from their graphical representation and hierarchical structure, which allows users with non-technical backgrounds to understand them. Indeed, one of the common uses of DFDs is in explaining the static structure of a system to non-technicians.

Traditional DFDs consist of bubbles and flows. Bubbles are drawn as circles and represent either processes (if the system being specified is concurrent) or procedures. Flows are drawn simply as arrows connecting the bubbles, and show the paths over which data may travel. Hence, a DFD is a directed graph. Flows coming into a bubble are called inflows, and flows leaving, outflows. A bubble reads the information on its inflows, and produces information on its outflows.

3.1.2 Motivation for Formalization of Traditional DFDs

While the description above is simplified, it is enough to point out the greatest flaw of traditional DFDs — there is no way to know when a bubble will read its inflows or produce on its outflows, and there is no way to know what a bubble will produce. This is partially explained by the fact that traditional DFDs are intended as static "road maps" of how information is transformed in a system, so that there was no notion of actually "executing" a DFD. This is clearly insufficient for precisely specifying a real software system.

A number of techniques have been used to combat this problem. In particular, the functionality of a bubble is often expressed in "Structured English" [You89]. This has the advantages of informality and some level of understandability by non-technicians, but is too ambiguous to be a good specification technique. Another technique is to include in the DFD the actual code implementing a bubble [GRW91]. While this is a formal specification, it is not an adequate specification technique, as it is too low-level. A third modification to DFDs describes the functionality of bubbles

with a finite state machine. This ends up as a notational convenience, for the finite state description is easily transformed into another traditional DFD.

A better technique is embodied in our language DFD-SPECS, a formal specification language based on DFDs. In DFD-SPECS, a set of *rules* is associated with each bubble. Each rule has three parts: an *enabling condition* that describes when a bubble may read its inputs, a *pre-condition* that gives conditions that must be met for the bubble to produce results, and a *post-condition*, which defines what the bubble outputs to its outflows. The enabling condition resembles the *when* clause of GCIL [Ler91]. These conditions are written as first order predicate calculus (FOPC) assertions over the values on the inflows and outflows of the bubble. Thus, DFD-SPECS has much in common with VDM [Jon90] and Z [Hay87] [Spi92] [Spi89] in that specification is done using FOPC pre- and post-conditions. DFD-SPECS thus has the formality of these specification techniques, and also the advantage of a graphical notation.

The traditional advantages of formality include terse, precise, and unbiased descriptions of functionality and the ability to verify properties of specifications. DFD-SPECS shares these advantages with other formal specification languages, and also uses this formality as the basis for *executing* specifications. Thus, most specifications can serve as prototypes of the final system, providing even those who are unfamiliar with the formal notation used the ability to experiment with and evaluate the specification. Specifications can also be tested and "debugged" in the same way that programs are debugged. This validation of specifications is important in the software development process, because it is usually much more cost effective to correct an error in the specification rather than later in the development process. For large projects, correcting an error in the maintenance phase is typically 100 times more costly than correcting the same error in the specification phase [Boe81].

Executablity for a large subset of DFD-SPECS is provided by an interpreter written in the functional programming language Standard ML [Pau91]. The interpreter also serves as a formal semantics for the executable subset of DFD-SPECS. None of the code for the interpreter is presented in this paper, but is available in [WBL93].

The rest of this paper is organized as follows. In Section 3.2 we introduce the

reader to the DFD-SPECS language. In Section 3.3 we briefly describe the technique used for evaluating enabling and pre-conditions, and for producing outflow values from post-conditions. Together, these sections characterize the execution behavior of DFD-SPECS specifications. In Section 3.4 we describe related work on the execution of DFDs and model-based formal specifications. In Section 3.5 we conclude with a description of the problems remaining in formalizing and executing DFD-SPECS.

3.2 Informal Description of DFD-SPECS

The description of the syntax and semantics of DFD-SPECS in this section follows that of Coleman [Col91].

3.2.1 Syntax of DFD-SPECS

A DFD-SPECS specification consists of a directed graph and an associated textual part. As in traditional DFDs, the nodes of the graph are the bubbles, and the arcs are the flows. Stores are simply flows with multiple origin and destination bubbles [LWBL92]. Each bubble is labeled with its name, and each flow with its name and type. Arcs with a double arrowhead are used for *persistent* flows, while single arrowheads are used for *consumable* flows. The difference between these kinds of flows is described in the next section.

The textual part consists of the data dictionary of types and abstract functions used in the specification, and the specifications of the bubbles (or *processes*), as described in the following paragraphs and extended BNF grammar.[2] The interpreter uses an abstract syntax that is basically a parse of this concrete syntax.

textual-part ::= [*data-dictionary*] *process**

The data dictionary defines the types and abstract functions used in the specification. The notation *type-expr*$^{*|}$ refers to union types, i.e. `type intOrReal = int | real;`.

[2] In this grammar, optional parts are enclosed in square brackets [ ], and the notation *expr*$^{*;}$ means a ; separated list of zero or more *expr*s. The *expr*$^{*|}$ and *expr**, notations have analogous meanings.

data-dictionary ::= `Data Dictionary :` *type-decl*$^{*;}$[;] *abstract-function*$^{*;}$[;]
type-decl ::= `type` *var-name* = *type-expr*
type-expr ::= `int` | `real` | `bool` | `string` | `signal` | `set of` *type-expr* | *type-expr**|
`sequence of` *type-expr* | `tuple of` (*param-decl**,) | *type-name*

Abstract functions are functions used only internally to the specification — that is, they can be used in enabling, pre-, and post-conditions, but are not available to users of the specified system. They are used to help in modularizing the specification, and to make it more readable.

abstract-function ::=`define` *absfun-name*(*param-decl**,) `as` *type-expr*
`such that` *FOPC-expr*
param-decl ::= *var-name* : *type-expr*

Each bubble is described by its name, initial state, and set of firing rules. The initial state specifies the initial values on the bubble's outflows. For flows with multiple source bubbles, the values specified must be consistent. The firing rules are the enabling condition, pre-, and post-condition triples discussed previously. In the enabling condition and initial state, the assertion +*flow-name* is true exactly when information is present on flow *flow-name*, while -*flow-name* is true when no information is present. An omitted pre-condition is equivalent to just `true`.

process ::= `Process` *bubble-name* : [*initial-state*] *rule*$^{*;}$[;]
initial-state ::= `initially` *flow-enabled-list* [/\ *FOPC-expr*]
rule ::= `enabled when` *enabling-condition* [`requires` *pre-condition*]
`ensures` *post-condition*
enabling-condition ::= `true` | *flow-enabled-list* [/\ *FOPC-expr*]
flow-enabled-list ::= [*flow-enabled-list* /\] *flow-enabled*
flow-enabled ::= +*flow-name* | -*flow-name*
pre-condition ::= *FOPC-expr*
post-condition ::= *FOPC-expr*

The first order predicate calculus used in DFD-SPECS specifications is augmented with operations on the built-in types. Unprimed flow names refer to the

values on inflows, while primed flow names (') refer to outflow values. For each field of a tuple, DFD-SPECS provides a function with the same name to extract that field from the tuple. The symbol - is used for both arithmetic subtraction and set difference, and || for concatenation of sequences. The `index` function provides array-like indexing into sequences, `header` returns all of its argument sequence except the last element, and `trailer` returns all of its argument sequence except the first element.

FOPC-expr ::= `true` | `false` | `not` *FOPC-expr* | *FOPC-expr* `/\` *FOPC-expr* |
FOPC-expr `\/` *FOPC-expr* | *FOPC-expr* `=>` *FOPC-expr* |
`\forall` *var-name* `:` *type-expr* [*FOPC-expr*] |
`\exists` *var-name* `:` *type-expr* [*FOPC-expr*] |
{*FOPC-expr* | *FOPC-expr*} | (*FOPC-expr*) `:` *type-expr* |
int-literal | *real-literal* | *string-literal* | *bool-literal* | *var-name* |
flow-name | *flow-name'* | *absfun-name*(*FOPC-expr**,) |
unary-op(*FOPC-expr*) | *FOPC-expr binary-op FOPC-expr* |
{*FOPC-expr**,} | <*FOPC-expr**,> | (*FOPC-expr**,) |
`index`(*FOPC-expr*, *FOPC-expr*)

unary-op ::= *field-name* | `size` | `first` | `header` | `last` | `trailer` | `length`

binary-op ::= `+` | `=` | `*` | `/` | `\mod` | `\union` | `\intersection` | `||` |
`=` | `<` | `<=` | `>` | `>=` | `\in` | `\subset` | `\subseteq` | `\supset` | `\supseteq`

As as example of a DFD-SPECS specification, consider the specification of a spelling checker in Figures 3.1 and 3.2. Roughly, process `FindWords` takes a document and the dictionary, and produces a dictionary of all the words in the document not appearing in the dictionary. The user of the specified system then supplies the correct spelling of each of these words. Finally, process `CorrectMisspellings` produces the corrected document and updates the dictionary. Flow `Dict: Dictionary` is an example of a store, while `OldDoc: Document` and `Unknown: Dictionary` are just diverging flows.

This specification has been successfully executed by our interpreter, and is a good illustration of the kinds of specifications that it can handle. For example, the specification makes use of both universal and existential quantification, and outflow values and abstract function results appear inside the bodies of quantified assertions.

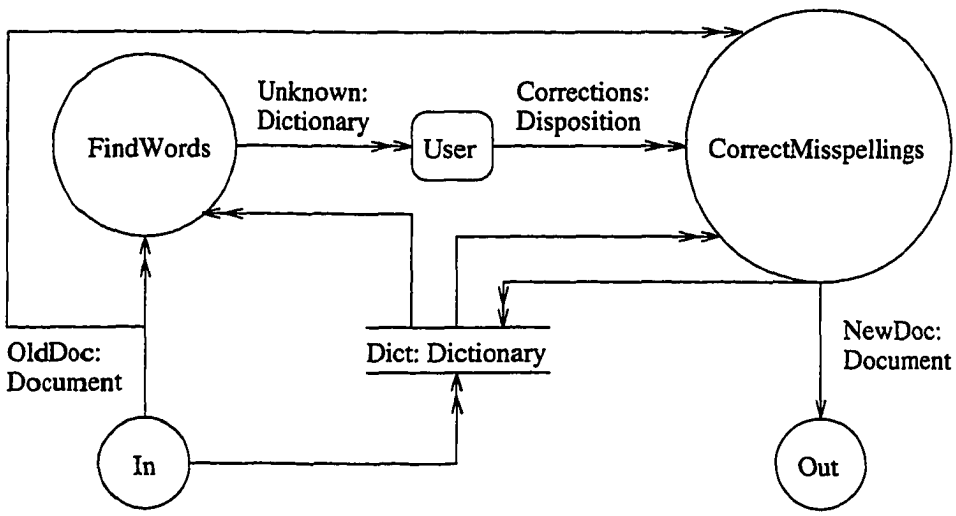

Figure 3.1: The graphical part of the DFD-SPECS specification of a spelling checker.

```
Data Dictionary:

 type Document = sequence of Word;
 type Word = string;
 type Dictionary = set of Word;
 type Disposition = set of WordPair;
 type WordPair = tuple of (UnknownSpelling: Word,
                           CorrectSpelling: Word)

 define HasCorrection(w: Word, C: Disposition) as bool such that
   IsKnown(w, C) = \exists c: WordPair [c \in C /\ w = UnknownSpelling(c)];

 define RightSpelling(w: Word, C: Disposition) as Word such that
   \exists c: WordPair [c \in C /\ w = UnknownSpelling(c)
                        /\ RightSpelling(w, C) = CorrectSpelling(c)]
```

Figure 3.2: Part 1 of the textual part of the specification of Figure 1.

```
Process In:

 initially +OldDoc /\ +Dict /\ OldDoc = <"teh", "quick", "browne", "fox">
          /\ Dict = {"brown", "dog", "the", "fox", "red", "slow", "quick"}

Process FindWords:

 initially -Unknown

 enabled when +OldDoc /\ +Dict
 ensures Unknown' = {w: Word | w \in OldDoc} - Dict

Process User:

 initially -Corrections

Process CorrectMisspellings:

 initially -NewDoc

 enabled when +OldDoc /\ +Corrections /\ +Dict /\ +Unknown
 requires \forall w: Word [w \in Unknown => HasCorrection(w, Corrections)]
          /\ |Unknown| = |Corrections|
 ensures \forall i: int [(1 <= i <= length(OldDoc)
                         /\ index(OldDoc, i) \in Dict)
                           => index(NewDoc', i) = index(OldDoc, i)]
      /\ \forall i: int [(1 <= i <= length(OldDoc)
                         /\ not (index(OldDoc, i) \in Dict))
                            => index(NewDoc', i) =
                                 RightSpelling(index(OldDoc, i),
                                               Corrections)]
      /\ length(NewDoc') = length(OldDoc)
      /\ Dict' = Dict \union
                 {CorrectSpelling(c: WordPair) | c \in Corrections}
```

Figure 3.3: Part 2 of the textual part of the specification of Figure 1.

Additionally, the sequence value that process `CorrectMisspellings` places on flow `NewDoc: Document` is defined in terms of `index` positions in the sequence, rather than directly. Section 3.4 contrasts this level of executability with that provided by other executable specification languages.

3.2.2 Informal Semantics of DFD-SPECS

This informal description of DFD-SPECS semantics as implemented by our interpreter follows the first formalization by Coleman [Col91] and the operational semantics given in Leavens et al. [LWBL92] for traditional DFDs. The key concept is that of *firing* a bubble. Firing is the process in which a bubble reads its inflows and produces onto its outflows. The semantics of firing gives meaning to the firing rules in DFD-SPECS, which specify the dynamic behaviour of a DFD.

We model how a bubble fires in two steps. A bubble first reads its input flows, and then writes to its output flows. We say a bubble is *working* when it has read its input flows but not yet produced output; it is *idle* otherwise. We consider the transitions between these states to be atomic.

What happens when a flow is read depends on the persistency of the flow. When a bubble reads from a consumable flow, the information read is removed from the flow, while reading from a persistent flow does not affect the information it contains. Similarly, writing to a consumable flow adds to the information on the flow, while writing to a persistent one overwrites any information already present.[3] Thus, a persistent flow is like a variable shared between two bubbles which the origin bubble can write and the destination bubble can only read. We treat consumable flows as unbounded FIFO queues. Hence, any information read from a consumable flow is read from the head of the flow's queue, and any information added to a consumable flow is added at the rear.

Firing occurs as follows. Initially, all the bubbles are idle. The flows may have some initial values placed upon them — the initial state of the diagram. Then the following algorithm is executed:

[3]The distinction between persistent and consumable flows is closely related to the issue of continuous versus discrete flows found in the traditional DFD literature [Col91] [HP87] [War86].

1. Find the set of bubbles that may fire. This includes all bubbles in the working state, and any bubble in the idle state that has values on its inflows satisfying the enabling condition of at least one of its firing rules.

2. Choose one of these bubbles to fire.

3. Fire the bubble:

   - If the bubble is idle:
     (a) Choose one of the bubble's rules whose enabling condition is satisfied by the inflow values.
     (b) Read the values referenced by this rule from the inflows. For consumable flows, remove the value. Otherwise, do not change the flow.
     (c) Change the state of the bubble from idle to working.
   - If the bubble is working:
     (a) Check that the pre-condition is satisfied by the values from the inflows. In the interpreter, an exception is raised if the pre-condition isn't satisfied.
     (b) Produce values onto the outflows. These values are defined by the post-condition of the rule chosen when the bubble changed to the working state. For consumable flows, the value is enqueued. For persistent flows, the new value overwrites the flow's contents.
     (c) Change the state of the bubble from working to idle.

4. Repeat the above steps until the set of bubbles allowed to fire in step one is empty.

So, in the example of the previous section, we start in a state where all bubbles are idle, and flows are initialized to the initial state. The only bubble that can fire is `FindWords`, because the enabling condition of its rule is satisfied and `CorrectMisspellings` has no rules with satisfied enabling conditions. Both of the inflows of `FindWords` are persistent, so they are read but not changed in any way. Finally, `FindWords` changes state to working.

Continuing the execution, `FindWords` is again the only bubble which may fire. Its rule has no pre-condition, which is equivalent to a pre-condition of just true and so is satisfied. Next, the dictionary of unknown words as defined by the post-condition is written to flow `Unknown: Dictionary`. Finally, `FindWords` changes state to idle.

3.3 Executing Firing Rules

This section briefly describes the technique used to execute the firing rules. The technique works with FOPC assertions over a fixed set of types (and a fixed set of primitive functions), and so is not dependent on DFDs. In fact, the execution technique has also been used in the execution of a specification language for C++ classes [WBL94]. More details on the execution technique are available in [WBL93], which presents part of the code for the interpreter.

As can be seen in the previous section, executing DFD-SPECS specifications requires the ability to use FOPC assertions in two distinct ways: to check whether a given set of values satisfies an assertion for testing enabling and pre-conditions, and to construct a set of values that satisfy an assertion for evaluating post-conditions.

Checking assertions is simpler, as all that is required in most cases is the straightforward implementation of logical and DFD-SPECS operators. The only possible complications are quantified assertions and set comprehensions (for example, `{x+1 | 1 <= x <= 5 /\ x mod 2 = 0}` is a set comprehension with value `{3, 6}`). In the executable subset, quantified variables and the bound variable in set comprehensions are required to range over a finite and explicit domain. For example, each variable bound by a quantifier in the example of Figures 3.1 and 3.2 is forced to range over a finite set of of values. Thus, quantified assertions are evaluated by simply iterating over this domain. For set comprehensions, the domain is filtered according to the predicate in the body of the comprehension, and then the result of this filtering is transformed (mapped, in functional programming terminology) according to the function in the head of the comprehension.

Finding values that satisfy post-condition assertions is considerably more complicated. The following algorithm is used to find outflow values that satisfy the post-condition of a firing rule.

1. Split the post-condition into two parts – the part that can be used in constructing outflow values, and the part that can only be used in checking that the values constructed satisfy the post-condition.

2. Evaluate the constructive parts into a list of constraints on the outflow values.

3. Evaluate the constraints into values for the outflows, and bind the outflow names to the appropriate values in an environment that can be used to check the nonconstructive parts of the post-condition.

4. Check that the nonconstructive parts of the postcondition are satisfied by the values constructed from the constructive parts.

To evaluate universally quantified assertions into constraints, the body of the assertion is evaluated once for each element of the domain quantified over. All constraints so produced are placed in the list of constraints generated from the post-condition. For existentially quantified assertions, the technique is to first find some element of the domain that satisfies the parts of the body of the assertion that do not refer to outflow values. Then, this element is bound to the quantified variable, and the body is evaluated once more to generate constraints.

3.4 Related Work

The work related to our research on executing DFD-SPECS falls into two categories: work on formalizing and executing DFDs, and work on executing model-based specifications. In the first category, the work that the authors are aware of requires the specifier to give at least part of the specification in a programming language, and thus the specification advantages of FOPC — abstractness, conciseness, freedom from implementation bias, and the possibility of using theorem proving techniques to prove properties of specifications, are lost. In the second category, existing techniqes for executing specifications are not suited to our purposes for various reasons, as detailed with the discussion of these techniques.

3.4.1 Formalizing and Executing DFDs

PAISley [ZS86] is an executable specification language based on asynchronous processes, which are similar to finite state machines. Thus, PAISley specifications have at least a superficial similarity to DFD-SPECS specifications, with states roughly playing the role of bubbles, and state transitions acting as flows. In PAISley, the computations that occur on state transitions are written in a functional programming language that features composition, conditional selection, and tuple formation.

PSDL [BL90] [LVY88] is an executable enhancement of DFDs for hard real-time specification. The language includes an interesting and elegant type system for flows. However, the atomic operations of the specified system — the operations that implement the functionality of bubbles — must be written in some programming language. Thus, both PAISley and PSDL sacrifice abstraction for executability.

3.4.2 Executing Model-based Specifications

The EPROL specification language [HI88, HI86] makes a fairly large subset of VDM executable by compiling it to LISP. Even some specifications that use quantifiers are executable. EPROL is part of the EPROS system, which provides tool support for evolutionary and functionality prototyping and for building the user interface to the prototype. The *me too* specification language [Hen86] executes a subset of VDM that is very similar to that executed by EPROL.

As VDM is closely related to the part of DFD-SPECS that is used for specifying enabling, pre- and post-conditions, either EPROL or *me too* could have been used in executing DFD-SPECS. However, neither language can execute assertions of the following three forms. These forms are crucial in writing sufficiently abstract assertions.

- The only operator that can be used to define post-state values[4] in EPROL and *me too* is =. In other words, the expression `S' = {3}` is executable, where `S'` represents the post-state value of the set `S`, but the equivalent

[4] In this context, post-state value is a synonym for outflow value — a value defined by the post-condition.

```
3 \in S' /\ |S'| = 1
```

is not.

- Post-state values of tuple, set, and sequence types cannot be defined "by parts", i.e. by describing the result of applying tuple, set, and sequence observers to such post-state values. For example, if rationals are modeled as a tuple consisting of `numer` and `denom` fields, then an assertion such as

  ```
  numer(r') = 3 /\ denom(r') = 4
  ```

  is not executable, even though it clearly defines the post-state value of `r`.

- A quantified assertion can only be evaluated if it contains no references to post-state values, and so can only be evaluated for its boolean value. Thus, an expression such as

  ```
  \forall x [x \in  S => x \in S']
  /\ \forall y [ y \in S' => y \in S]
  ```

  is not executable, even though it is clearly equivalent to `S' = S`, which is executable using any of the techniques described.

All of the above assertions are executable in DFD-SPECS.

Another approach to the execution of model-oriented specifications is the **fase3** language of Kamin and Kraus [KK93, Kra88]. Only a few primitive types such as integer, boolean, and so on are built into **fase3**. More structured types, such as the set, sequence, and tuple types of DFD-SPECS are specified using a style of final algebra specification that allows any type to be represented as a tuple of functions. As long as these functions remain finite for a particular element of the type, the functions can be represented as "tables" (finite sets of tuples), and quantified assertions over that element can be executed. Unlike quantified assertions in executable DFD-SPECS specifications, the specifier need not supply an explicit bound on the quantified variable. Assertions that use observers to define values are also executable

— in fact, the functions that represent an element are exactly the observers of the type as applied to that element.

However, many quantified assertions that are executable using our technique for DFD-SPECS can not be executed in **fase3**. In **fase3**, most quantified assertions with references to the specified function's return value in their bodies are not executable[5]. The only exception is a restricted form of existentially quantified assertions that is used to select an element from the domain quantified over. Thus, for example, the third example of a specification that EPROL cannot execute (given above) is also an example of a specification that is not executable in **fase3**.

The execution techniques are quite different. In **fase3**, specifications are first compiled to an extended λ-calculus, and then to an extended form of combinator graph, which is then reduced. An execution technique of this generality seems to be necessary because the specifier defines the observers of a particular type, rather than the observers being built into the language as in DFD-SPECS. It seems likely that errors reported by the reduction algorithm will not be as useful in debugging specifications as errors reported by a direct interpreter such as the one used for DFD-SPECS, as the combinator graph that the reduction algorithm is working on is several compilation steps removed from the specification that the graph was derived from.

3.5 Conclusion

3.5.1 Contributions

We wish to emphasize four contributions of this research. Two pertain directly to the specification of DFDs, the third highlights the advantages of having a formal semantics for DFD-SPECS, and the fourth presents the use of executable specifications in software development.

The first contribution to the specification of DFDs is the particularly simple structure of enabling conditions, as compared to other efforts to capture this idea [Kun91] [Col91]. In our effort to execute and give semantics to enabling conditions, we found that it does not make sense to allow arbitrary logical connectives in the

[5]Note that the return values of **fase3** functions are the values that are defined by specifications, and so play the role of outflow values in DFD-SPECS.

flow-enabled-list portion of the enabling condition (the part consisting of the list of inflows of the bubble tagged with + or -; see the grammar in Section 3.2.1). This realization occurred while automating evaluation of enabling conditions, and so is a result of our effort to execute DFD-SPECS. Consider, for example, an enabling condition (that we do not allow) of the form `+f1 \/ +f2`. This enabling condition is true when a token is present on either flow, or on both. Thus, when the rest of the rule is written, there is no way to know whether or not any individual flow has a token on it, and so no other part of the rule may refer to the value of a token consumed from either flow. The information carried on the flows is lost. So, the only logical connective for the elements of the *flow-enabled-list* allowed in DFD-SPECS is /\.

The other contribution to the specification of DFDs is an insight into the nature of firing rules. By adding the notion of firing a bubble, we have raised the questions: "Can a bubble fire concurrently with itself? Can a bubble fire on two distinct rules simultaneously?" In the execution of DFD-SPECS, the answer is no — the semantics we have given prohibits this behavior. When a bubble in the working state fires, it must fire on the rule it was using when the transition to the working state was made, and must produce its output and change state to idle. Work on refinement notions for DFD-SPECS [Lyl92] has shown that if a bubble is refined to a sub-diagram with several bubbles, the sub-diagram can exhibit behaviors that look like reading more than one input before producing output. We have investigated formal semantics for bubbles firing concurrently with themselves in a forthcoming work [LWBL92], and the application of this work to DFD-SPECS would be straightforward.

By formalizing the DFD model and providing a semantics for DFD-SPECS, we have made the rather "warm and fuzzy" Data Flow Diagram into a solid tool for specifying software systems. Thus, we have all the advantages of traditional formal specification, and at the same time, our model still allows users to do traditional DFDs — simply by building a diagram with bubbles and flows, but no rules, or even simplified rules that don't capture the entire functionality needed. This informal specification can be directly refined to a formal one — by adding the necessary rules [Lyl92].

The final contribution of our work is that most DFD-SPECS specifications are

executable. Thus, the specification is a prototype of the final system, allowing conceptual and requirements errors to be found before the lengthy and expensive process of coding begins. Additionally, the specification can be tested and debugged, allowing errors in the specification can be found and corrected before valuable programmer time is put into implementing the specification. Both these points imply that errors are likely to be discovered earlier than with a traditional software development cycle, resulting in quicker and less expensive production of software.

3.5.2 Directions for further research

We have formalized and made executable much of the traditional DFD model, but more remains to be done. Some of what follows arises from traditional DFDs, but much of the research is unique to the formal model.

3.5.2.1 Research arising from traditional DFDs An interesting possibility stemming from our research is that of using our formal semantics with traditional CASE tools. In particular, Teamwork^{tm} has already been modified to work with DFD-SPECS specifications. We would like to combine our interpreter with Teamwork to produce a tool that could animate the graphical representation of DFD-SPECS.

3.5.2.2 Research arising from DFD-SPECS The problems we have uncovered in formalizing DFDs are perhaps even more interesting than those found in traditional DFDs. We have extended a static model to a dynamic one, and so face issues that don't even exist in the traditional DFD world.

For example, we would like to write *invariants* for DFD-SPECS specifications. An invariant is a logical assertion defining allowable states of the diagram, and so only makes sense in terms of a dynamic model. For example, we might like to require that a certain flow never be empty, or that it never contain more than some fixed number of tokens — consider specifying a bounded buffer producer/consumer system. Since our specifications are now executable, we would now also like to check that the invariant holds at every step in the computation of the diagram. Extending the

idea of invariants, we may eventually want to use temporal logic [MP92] [Pnu86] to describe liveness and safety properties of DFD-SPECS specifications.

We would also like to extend the subset of FOPC that can be used in executable DFD-SPECS assertions. We are studying additional techniques for enhancing the interpreter, such as backtracking and running our execution technique multiple times on a single post-condition for finding outflow values that are defined in terms of other outflow values.

3.5.3 Concluding Remarks

Thus, it is with high expectations that we approach the next stages of our research. We have successfully demonstrated the executability of a large and powerful subset of DFD-SPECS. With the addition of a parser to translate the concrete syntax of DFD-SPECS to our abstract syntax, our interpreter could become a useful tool in industrial software development settings.

4. AN APPROACH TO THE DIRECT EXECUTION OF MODEL-BASED SPECIFICATIONS

A paper in preparation for the Software Engineering Journal

Tim Wahls,[1] Albert L. Baker, and Gary T. Leavens

Abstract

Executable specification languages may be the key to more widespread use of formal methods in software production. However, the expressiveness of executable specification languages is typically much less than that of non-executable specification languages such as VDM or Z, and so specifiers are forced to work at a lower level of abstraction to gain the advantage of executability. Additionally, specifications are typically made executable by translating them to some programming language, so many errors in the specification can only be detected as errors in the resulting code. This paper presents a technique for directly executing model-based specifications written at a relatively high level of abstraction.

4.1 Introduction

One barrier preventing formal methods from playing a larger role in industrial software development is the fear that the benefits of using formal methods are not worth the costs. Executable formal specifications can help in overcoming this fear, as an executable specification yields an immediate prototype of the final system.

[1]Wahls's work is supported by a fellowship provided by IBM Rochester. Leavens's work is supported in part by the National Science Foundation under Grant CCR-9108654.

Those who already use formal specifications also benefit, because executing a specification allows debugging it in the same way that programs are debugged. In addition, executable specifications ease incremental system integration and testing, as the specification (as a prototype) can take the place of parts of the final system that are yet to be implemented.

Thus it is not surprising that many executable specification languages have been developed. Unfortunately, these languages almost always force the specifier to work at a much lower level of abstraction than is normally employed in writing specifications in non-executable model-based specification languages, such as VDM [Jon90] and Z [Hay87, Spi88, Spi89, Spi92]. Typically, either the specifier must provide the algorithms needed for executing the specification [LL91, BL90, ZS86, LB89], or execution is based on translating the specification to Prolog [TC89, WE92, DKC90]. In the former case, the specifier is forced to work in an impoverished specification language that is often almost indistinguishable from modern functional programming languages, and the algorithmic information that must be provided is a potential source of implementation bias [Jon90]. If the specification is translated to Prolog, the person using the executable specification is exposed to the inefficiency and non-logical features of Prolog (cut, dependence on order of clauses). Such translations also usually entail poor reporting of errors. These problems are magnified when validating specifications, as specification errors show up in the Prolog code, and the validator must first find the error in the context of the Prolog code, and then find the corresponding error in the specification.

A number of existing executable specification languages/techniques allow the execution of implicit specifications — that is, specifications that do not explicitly provide the necessary algorithms. We first briefly describe several of these languages, and then discuss some of the limitations that these languages share.

The EPROL specification language [HI88, HI86] makes a fairly large subset of VDM executable by compiling specifications to LISP. Even some specifications that use quantifiers are executable. EPROL is part of the EPROS system, which provides tool support for evolutionary and functionality prototyping and for building the user interface to the prototype. Similarly, the *me too* specification language [Hen86] executes much the same subset of VDM by embedding specifications in LISP.

The UK's National Physical Laboratory has developed a syntax-directed editor for entering VDM specifications with a mode called *SMLVIEW* [O'N92a, O'N92b] that displays the VDM specification entered compiled into Standard ML (SML) [Pau91] code. The translation scheme is apparently fairly direct, so the VDM specification must be somewhat close to an SML program for this translation to work, and the subset of VDM that can be translated is relatively small. Additionally, some VDM features that are in principle executable, such as set comprehensions[2], are not implemented.

As both EPROL and *SMLVIEW* involve translation to a programming language, both force the user wishing to validate specifications into the "debug the code, then debug the specification" mode described previously, especially if the generated code fails to compile (as can happen when using *SMLVIEW*) or causes a run-time error. Any specification error discovered is then reported in the context of the generated code, rather than the context of the specification. As an EPROL specification can be written at a more abstract level than a specification to be executed using *SMLVIEW*, the distance between the specification and the Lisp code generated is relatively large, and the difficulty of finding the flaw in the specification that caused the error in the generated code may be great. As specifications to be executed using *SMLVIEW* must be fairly close to the resulting SML code, this "reverse translation" problem is likely to be less serious. However, with *SMLVIEW*, the user must often hand edit the SML code resulting from the translation to achieve even a syntactically correct SML program. In either case, then, the user is forced into intimate contact with a language that is not directly involved in the software development process – it is neither the specification nor implementation language.

Additionally, all three of the techniques discussed apparently cannot execute VDM-like assertions of the following three forms:

- The only operator that can be used to define post-state values is $=$. In other words, the assertion $3 \in S'$, where S' represents the post-state value of the set S, is of no help in building a post-state value for S, even though it clearly

[2]For example, $\{i+2 \mid 1 \leq i \leq 5 \wedge i \bmod 2 = 0\}$ is a set comprehension with value $\{4,6\}$

defines part of that value.

- Post-state values of tuple, set, and sequence types cannot be defined "by parts", i.e. by describing the result of applying tuple, set, and sequence observers to such post-state values. For example, if rationals are modeled as a tuple consisting of *numer* and *denom* fields, then an assertion such as $numer(r') = 3 \wedge denom(r') = 4$ is not executable, even though it clearly defines the post-state value of r.

- A quantified assertion can only be evaluated if it contains no references to post-state values, and so can only be evaluated for its boolean value. Thus, an expression such as $\forall x[x \in S \wedge odd(x) \Rightarrow x \in S']$ can not be used in constructing the value of S', even though it again defines part of that value.

In the case of *SMLVIEW*, these limitations can be seen in the description of the subset of VDM that can be executed (see Appendix J of [O'N92a]). Although the authors have been unable to locate any similar characterization of the subset of VDM that EPROL or *me too* can execute, the example specifications that have been published also exhibit these same limitations.

We present a technique that makes constructive use of these kinds of assertions in constructing post-state values, and that executes the kinds of assertions that can be executed by EPROL, *me too*, and *SMLVIEW* as well. While adding executability of these three features may seem like small progress, they are characteristic (in our experience) of the truly implicit specifications many specifiers, regardless of their expertise and mathematical sophistication, tend to write. Thus, the executability of these features is of key importance in executing specifications. We want executability to be as small a burden on the specifier as possible. Our main goal is to allow the specifier to write purely implicit specifications in a language that is in general not executable, and then execute as much of the specification as is practical. If more executability is needed, then the specification can be refined to an executable form. Thus, specification languages that are designed to be executable are in general not suitable for our purposes, but we do want to ease the trade-off between implicitness and executability as much as we can.

In addition, our technique has been implemented as an interpreter for model-based specifications, and so avoids the problems associated with validating a specification when translation to some programming language is used for execution. The technique is described in the context of the specification language SPECS-C++, which is a VDM-like language specialized for specifying C++ [ES90b, Str91] classes. However, our execution technique is not tied to any special feature of SPECS-C++, and so would work as well for VDM specifications as it does for those written in SPECS-C++.

An informal description of SPECS-C++ is presented in the next section. Our approach to executing SPECS-C++ follows in Section 4.3. Section 4.4 presents an informal characterization of the limitations of our execution technique. Finally, we examine some of the most closely related work in Section 4.5 and speculate about the continued evolution of directly executable, abstract, and formal specifications for C++ classes in the concluding Section 4.6.

4.2 Syntax for Executable SPECS-C++ Specifications

The next two sections summarize our work on direct execution of model-based specifications. We have a literate[3] Standard ML (SML) [Pau91] implementation of an interpreter for a subset of SPECS-C++. This SML implementation should be thought of as a prototype and a demonstration of the feasibility of the execution technique. as it uses an abstract syntax for SPECS-C++ that is too cumbersome for actually writing specifications. Other members of the SPECS-C++ development team are currently working on tools that use the concrete syntax of SPECS-C++ presented in this section. The next section describes the executable subset of SPECS-C++ and the execution technique used in the SML implementation. In the interest of brevity, no SML code is presented, but is available in [WBL94].

A SPECS-C++ specification consists of a set of class specifications and the

[3]The interpreter is being developed with the aid of the literate programming tool Noweb [Ram91]. Noweb allows a program and the text describing it to be written simultaneously and in the same file, provides a structured way of presenting the code, and makes extracting the code or the descriptive text easy.

definitions of the abstract model types used in those specifications. The primitive abstract types are basically those of VDM: `integer`, `float`, `char`, `string`, and `bool`, where `float` is the type of real numbers. The `void` type of C++ is included as the return type of pure procedures. More complex types are composed from these basic types: finite sets, sequences, tuples, and alternative (union) types. The map and function types of VDM are not included, but finite maps and functions may be modeled as sets of tuples. All abstract type definitions are visible globally.

Class specifications in SPECS-C++ roughly correspond to modules in VDM. Each class specification defines a type (with the same name as the class) and a set of operations on that type, called public member functions. These public member functions correspond to the exported operations of a VDM module, and are specified using pre- and post-conditions, much like implicit VDM specifications. However, the precise meaning of a class specification is different than the meaning of a VDM module, because SPECS-C++ is an *interface* specification language [GHW85, GHG+93, Lam89] for C++. That is, a SPECS-C++ specification of a class can only be implemented by a C++ class with the same name, each prototype of a public member function in the specification must appear (with the same name, return type, and arguments) in the implementation as a public member function, and the implementation of each public member function must satisfy the specification given for it in the SPECS-C++ class. For example, consider a SPECS-C++ class `OrderedPair` (representing ordered pairs of integers) and a member function `First` that returns the first element of an `OrderedPair`. (This specification will be presented in full shortly.) The relationship between the specification and the implementation is pictured in Figure 4.1. The interface and the meaning are both part of the specification that must be matched by the implementation.

As in Larch/Ada [GMP92] and LM3 [Jon92], SPECS-C++ specifications are embedded in implementation code. A SPECS-C++ specification is placed in specially formatted comments within a C++ header file. This automatically makes the interfaces match, and avoids redundancy. Thus, the declaration of the class and the prototypes of the member functions in the specification must be compilable C++ code, and the other parts of the specification (i.e., abstract type declarations, pre- and post-conditions, and so on) are written inside C-style (/* to */) comments, and

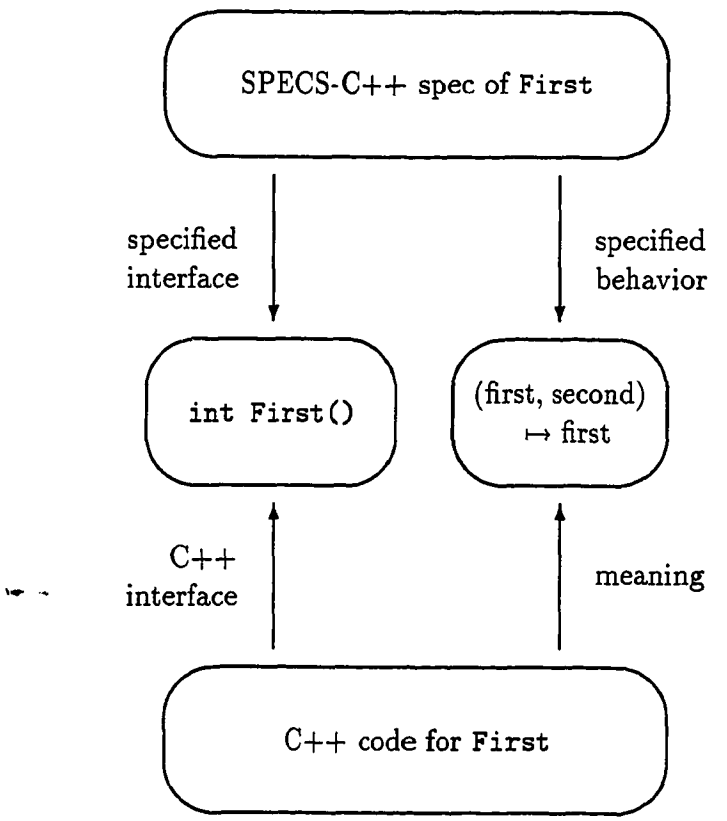

Figure 4.1: The meaning of a SPECS-C++ member function specification. The C++ code must match the specified interface and have a compatible meaning.

this style of comment cannot be used otherwise. This separates the C++ clients of the specified class from the C++ code that implements it, so that the implementation can be separately compiled. As the specification is included in the header, the header contains all the information that C++ clients of the class need to know to use the class correctly.

As an example of a SPECS-C++ specification, consider the specification of the classes `OrderedPair` (Figure 4.2) and `Relation` (Figure 4.3), where `Relation` is modeled as a set of `OrderedPairs`. Member functions with the class name are constructors, and are typically invoked in client code when class instance declarations are executed. Class `Int_Set`, the return type of `RelTo`, is not shown here as it is uninteresting, but would have to be included for executing this example. The pre-condition of the function `RelTo` specifies that the parameter `key` must be in the domain of the relation, and the post-condition specifies that the result consists of all the integers related to `key` in the relation. The obvious VDM analog of `RelTo`'s post-condition exhibits two of the kinds of assertions that are not executable under the VDM-based techniques discussed earlier, as references to post-state values appear inside quantified assertions, and the value of `result'` is specified using only `\in` (set membership). However, this post-condition is executable using our approach. As this post-condition is also a non-trivial example of the expressive power of these features, the specification of `RelTo` will be used as a running example in the section on executing assertions (Section 4.3). A more detailed description of SPECS-C++ is presented after this example.

In SPECS-C++, a class specification consists of the abstract data members, an invariant, a set of abstract function definitions, and a set of public member function specifications. Each of these parts is described below.

In C++, the data members of a class implement the state of instances of the class – i.e., elements of the type defined by the class. Abstract data members in SPECS-C++ differ from the concrete data members of C++ in two important ways:

- abstract data members are used to model the state of instances of the class type (and so appear only in the specification), while C++ data members actually implement these states. Hence, many different collections of concrete C++ data members can implement the specified abstract data members.

```
class OrderedPair {
/* model
**   data members
**     int first
**     int second
** operations
*/
public:

OrderedPair(int f, int s);
/* modifies: self
** postA: first' = f /\ second' = s
*/

int First();
/* postA: result' = first
*/

int Second();
/* postA: result' = second
*/
};
```

Figure 4.2: Specification of class `OrderedPair`.

```
class Relation {
/* model
**   data members
**     set of OrderedPair theRel
** operations
*/
public:

Relation();
/* modifies: self
** postA: theRel' = {}
*/

Insert(OrderedPair elem);
/* modifies: self
** postA: theRel' = theRel \union {elem}
*/

...

Int_Set RelTo(int key);
/* preA: \exists (OrderedPair p) [
**         (p \in theRel) /\ p.First() = key]
** postA: \forall (OrderedPair p) [
**          (p \in theRel) /\ p.First() = key => p.Second() \in result'
**        ] /\
**        \forall (int i) [
**          (i \in result') =>
**            \exists (OrderedPair p) [
**              (p \in theRel) /\ i = p.Second() /\ key = p.First()]]
*/
}
```

Figure 4.3: Specification of class Relation.

- abstract data members are visible globally in the specification, while C++ allows various kinds of control over what parts of a program may access concrete data members. The development method used with SPECS-C++ requires that the C++ data members implementing any given abstract data members be private (visible only within the class where they are defined). Thus, class implementations that satisfy a SPECS-C++ specification have only private data members, and all client code access must be through the specified public member functions.

In our semantics for SPECS-C++, class instances are modeled as tuples composed of the abstract data members, and so are similar to VDM composite types (trees). If a class has no abstract data members, then its type is modeled as the empty tuple. So, for example, instances of class `OrderedPair` would be modeled as follows:

```
tuple OrderedPair (int first, int second)
```

The invariant in SPECS-C++ is a first order predicate calculus assertion that every class instance must satisfy in every state that can be observed by client code. Invariants may also be written for any specifier-defined abstract type. In our example specifications, any pair of integers is a valid `OrderedPair`, and any set of `OrderedPairs` is a valid `Relation`, and so their invariants are both `true`, and hence omitted. The invariant for a class should refer only to the abstract data members of the class it belongs to.[4]

Abstract functions allow a first order assertion on the abstract model to be abstracted and parameterized, much like the functions of an ordinary programming language. Abstract functions cannot be invoked in client code. They are just aids in modularizing other assertions in a specification.

The specification of a member function consists of a C++ function prototype, a pre-condition, a `modifies` clause, and a post-condition. The function prototype includes a return type, the name of the function, and a parameter list with the types

[4]This restriction does not apply in the presence of specification inheritance. As specification inheritance can not be used (yet) in the executable subset of SPECS-C++, it is not discussed further here.

and names of the formal parameters, exactly as in C++. Member functions always take a formal parameter of the associated class type which is not explicitly listed, the *default parameter*. In SPECS-C++, this default parameter is always referred to as `self`, and is analogous to `*this` in C++. As a shorthand, the abstract data members of the default parameter can be referred to without mentioning `self`, so `first` and `self.first` are synonyms when specifying the member functions of class `OrderedPair`.

The `modifies` clause is a list of the formal parameters, possibly including the default parameter, that the member function is allowed to mutate. The `modifies` clause could also be used to advertise mutation of non-local objects, but this is not currently allowed by the SPECS-C++ method. An omitted `modifies` clause means that the member function has no side effects.

Just as in VDM, the pre- and post-conditions describe respectively what must be true for the member function to execute correctly, and what will be true after executing the member function. An omitted pre-condition is equivalent to just `true`, and means that successful execution of the member function does not depend on the pre-state. We take the total correctness approach, so an implementation is required to terminate in a state that satisfies the post-condition whenever the pre-state satisfies the pre-condition. In the post-condition, the keyword `result'` is used to refer to the return value of the member function. To distinguish between the pre-state and post-state values of formal parameters in the post-condition, variable identifiers representing post-state values are primed ('). We include `result'` in the set of primed identifiers. Primed identifiers may not appear in a pre-condition.

Pre- and post-conditions, abstract functions, and invariants are first order predicate calculus assertions written over SPECS-C++ types. Thus, SPECS-C++ expressions include the literals of the primitive abstract types, the normal mathematical operations on integers and floats, and the standard constructors and observers for sets, sequences, and tuples. Equality is defined in the natural way for each of these types. Sets are constructed by the standard `{...}` notation, including set comprehensions, i.e. `{i + 2 | 1 <= i <= 10}`. They are observed by `\in` and `\subset`, which are just membership and subset, respectively. Sequences are constructed by `<...>`, and `||` is used for appending of sequences. Given this much notation, the observers

that decompose sequences are defined as follows, where s represents a sequence, and e is a sequence element:

```
first(<e>||s) = e
last(s||<e>) = e
header(s||<e>) = s
trailer(<e>||s) = s
index(<e>||s, i) = if i = 1 then e else index(s, i - 1)
```

Note that none of these observers may be applied to empty sequences. The only built-in way to observe tuples is to look at their components, or fields. For each field of every tuple type, SPECS-C++ provides an observer with the same name as the field that returns the associated value. For example, given a declaration:

```
tuple rational (int num, int denom)
```

functions `num: rational -> int` and `denom: rational -> int` are then available for extracting the components of the tuple.

Predicate calculus assertions are built with the standard boolean connectives: logical and (`/\`), or (`\/`), implication (`=>`), and negation (`!`), and the universal (`\forall`) and existential (`\exists`) quantifiers.

Abstract function references and calls of member functions with non-void return types are also allowed in assertions. Any side effects of the called member function are ignored, so the meaning is that the value[5] defined by the body of the abstract function, or respectively the value of `result'` defined by the post-condition of the member function, is substituted for the call or reference in the assertion. (A syntax and informal semantics for using side effects of member functions in assertions has been developed, but this feature is yet to be formalized.)

---

[5]While the specifications of member functions can in general be nondeterministic, we require abstract functions and member functions that are referenced in pre- or post-conditions to be deterministic. Otherwise, the semantics of assertions becomes problematic. For example, consider the truth value of the assertion `foo(i) = 3` when `foo(i)` can evaluate to either 2 or 3.

4.3 An Algorithm for Executing Assertions

Only public member functions of SPECS-C++ classes can be referenced in client code, so the direct execution of SPECS-C++ (from a client's perspective) is just the execution of public member function calls.[6] A call to `RelTo` would have the form `R.RelTo(2)`, where `R` is an instance of class `Relation`. Note that the parameters are not C++ values, but rather abstract values corresponding to C++ values. To interpret such a SPECS-C++ public member function call, we use its specification as follows:

1. Check that the actual arguments satisfy the pre-condition of the called member function.

2. Construct post-state values that satisfy the post-condition of the called member function.

If the post-condition isn't satisfiable, or if the execution technique isn't adequate for the given post-condition, then this attempt will fail.

4.3.1 Evaluating Pre-conditions

As the pre-condition is evaluated in the pre-state of the operation, all the values in the pre-condition assertion are known – they are exactly the actual arguments of the member function. Thus, all that is needed is to apply the definitions of the built-in operators, and the implementation is straightforward. The quantified assertions and set comprehension expressions are the only possible complications.

In the executable subset of SPECS-C++, universally quantified assertions must be of the form:

```
\forall (T x) [ (BP(x)) /\ P(x) => Q(x) ]
```

and existentially quantified assertions must be of the form:

[6] Invariants could also be executed in order to check that all class instances created or modified by member functions satisfy their respective invariants. However, we have not done so to-date.

```
\exists (T x) [ (BP(x)) /\ P(x) ]
```

where T is the type of the bound variable, and `BP(x)` is either `x \in E` and E is a finite set or sequence, or `BP(x)` is `low <= x <= high`, for some integer valued expressions `low` and `high`. We use the term *domain* of the quantified variable to mean either the predicate `BP(x)` or the set of values that satisfy that predicate. Which meaning is intended should be clear from context. In the executable subset, this forces quantified variables to range over finite domains. The assertions `P(x)` and `Q(x)` are arbitrary assertions of the executable subset (including quantified assertions) that may refer to `x`, and the `/\ P(x)` portion of both quantifier forms is optional.

A universally quantified assertion such as:

```
\forall (int x) [(1 <= x <= 5) => x < 6 ]
```

is evaluated by applying the predicate `x < 6` to each of the integers 1 through 5, and logically "anding" together each of the results. Thus, evaluating universal quantification corresponds to an *and reduction* over the domain of the quantified variable. For existential quantifiers, an assertion such as the pre-condition of `RelTo`:

```
\exists (OrderedPair p) [(p \in theRel) /\ p.First() = key ]
```

is evaluated by applying the predicate `p.First() = key` to each `OrderedPair` in `theRel`, and logically "oring" together all of the results – which corresponds to an *or reduction*. This technique of evaluating quantifiers with reductions is common in the literature [BM93].

In the executable subset, set comprehensions must be of the form:

```
{F(x) | (BP(x)) /\ P(x)}
```

where `BP(x)` and `P(x)` are used in precisely the same manner as they were in the definitions of the executable quantified assertions. The term `F(x)` is an arbitrary term of the executable subset of SPECS-C++. The term `F(x)` is not required to contain any occurrence of `x`, but will in any interesting case. Given this definition, the evaluation of a set comprehension is basically a filter of the domain, followed by a map. For example, the expression:

```
{x + 2 |(1 <= x <= 5) /\ x mod 2 = 0}
```

is evaluated by choosing the integers between 1 and 5 that are even, adding 2 to each, and making a set out of the results.

4.3.2 Satisfying Post-conditions

Executing a post-condition is more involved. The following steps provide an overview of our algorithm for executing a post-condition:

1. Split the post-condition into constructive and nonconstructive parts. The constructive parts are those which will actually be helpful in constructing post-state values that satisfy the post-condition.

2. Generate a list of constraints from the constructive part. Each constraint defines a post-state value or some part of a post-state value.

3. Solve the constraints to construct the portion of the post-state that differs from the pre-state. The constraints need not determine unique post-state values (and so some "looseness" in executable specifications is possible), but this process is deterministic. Thus, calling the same sequence of member functions with the same arguments will always produce the same results.

4. Check the nonconstructive portion of the post-condition by evaluating it in a state where unprimed identifiers have their pre-state values, and primed identifiers (including `result'`) have their post-state values. This state is built from the pre-state and the state constructed in the previous step. If this check fails, then the execution of the post-condition fails.

5. Construct the state that results from the member function call. This state reflects mutations of the actual parameters of the call.

We are planning to generalize our algorithm by supporting backtracking; if the evaluation fails in step three or step four, the algorithm could go back to step one and make a different split into constructive and non-constructive parts, or to step three and try some of the other possibilities that any looseness in the specification permits.

We will discuss the possibilities for backtracking in more detail as the algorithm unfolds.

4.3.2.1 Splitting into Constructive and Nonconstructive Parts The first step is to split the post-condition into two parts: the part that will be useful in constructing post-state values, and the part that can only be used as a check on post-state values. This splitting is based on the structure of the post-condition. For example, in an assertion of the form `A1 /\ A2`, both `A1` and `A2` must be true for the post-condition to be satisfied, and so each is processed separately.

With an assertion of the form `A1 \/ A2`, the splitting algorithm tries to determine if either `A1` or `A2` is necessarily false, just using pre-state values. If either is necessarily false, it discards that argument and continues processing the other. For example, in the post-condition: `(x = 3 /\ x' = 4) \/ (x != 3 /\ x' = 3)` (recall that primed variable identifiers represent post-state values), one argument of the `\/` assertion will be false just from pre-state values, and so can be ignored. If neither argument can be determined to be false, then the entire `\/` assertion is nonconstructive, and is evaluated only after the construction of all post-state values. For example, the entire assertion: `x' = 3 \/ x' = 4` is nonconstructive. However, when backtracking is added to the interpreter, this case can be handled differently. The interpreter can assume that `x' = 3` is true, and proceed with the execution. If this assumption results in an unsatisfiable or non-executable post-condition, then the interpreter can backtrack and make the other possible assumption (`x' = 4`). If this assumption fails, then the post-condition must be unsatisfiable or non-executable.

The first step in evaluating an assertion of the form `A1 => A2` is to determine whether the antecedent `A1` is necessarily true or false, in the same way that the arguments of an `\/` assertion are tested. If `A1` must be false, then the entire `=>` assertion is ignored, as there is no requirement that the consequent `A2` holds. If `A1` must be true, then `A2` must hold, and so is processed recursively. If the truth value of the antecedent cannot be determined just from the pre-state values, then the entire assertion is nonconstructive. Again, the addition of backtracking allows better handling of this case. The two possible assumptions that can be made are that `A1` and `A2` are both true, or that `A1` is false.

Of the post-condition operators that are not logical connectives, the most important are =, `\in`, and `\subset`. These are the ones that directly contribute to the construction of post-state values. For one of these operators to be constructive, it must be the case that one argument of = and the left argument of both `\in` and `\subset` contain no primed identifiers. This allows the value of that argument to be computed in the pre-state in much the same way that pre-conditions are evaluated. The other argument of =, and the right argument of both `\in` and `\subset`, must represent a post-state value or part of a post-state value. In other words, it needs to be a primed identifier, or arbitrarily many applications of the built-in tuple and/or sequence observers to a single primed identifier. (The tuple and sequence observers were discussed in Section 4.2.) For example, `x' = 3`, `elem \in theRel'`, and `{3} \subset result'` are all constructive, but `result' = self'.Second()`, `5 * x' = 3`, `x' in {3, 4}` and `index(S', i') = 3` are not. Clearly, "constructive" information can be obtained from assertions that are not currently constructive, and so this is an area of ongoing research. In Section 4.4, we discuss an iterative technique for relaxing these restrictions.

Other kinds of relational operators and negated assertions provide some information about post-state values, but are usually of little help in actually constructing them, and so are classified as nonconstructive. For example, `x' < 3` or `!(x' = 3)` do give some information about the post-state value of `x`, but both are satisfied by an infinite number of post-state values. One possible extension of the work presented here is to use constraint-satisfaction techniques [Lel88] both for gleaning more information from these kinds of expressions and for generalizing what can represent a post-state value in a constraint.

Set comprehensions, abstract function references, and calls to member functions are constructive if they contain no occurrences of primed identifiers. For a universally quantified assertion to be constructive, the expression for the (finite) domain that the quantified variable ranges over and the assertion `P` (the rest of the antecedent of the implication) must not contain primed identifiers. Additionally, the assertion `Q` (the consequent of the implication) must be constructive, using the definition being developed in this section. Otherwise, the universally quantified assertion is nonconstructive. For example, in the post-condition of `RelTo`:

```
\forall (OrderedPair p) [
  (p \in theRel) /\ p.First() = key => p.Second() \in result']
```

is constructive, as neither the domain of the quantified variable nor the antecedent of the implication contains a primed identifier, and the consequent of the implication is constructive. However,

```
\forall (int i) [
  (i \in result') =>
    \exists (OrderedPair p) [
      (p \in theRel) /\ i = p.Second() /\ key = p.First()]]
```

is not constructive, because `result'` is the domain of the quantified variable.

Existentially quantified assertions are constructive if the domain of the bound variable contains no primed identifiers, and the rest of the assertion is constructive.

4.3.2.2 Evaluating Constructive Parts into Constraints To simplify the construction of post-state values, the constructive part of the post-condition is transformed into a list of constraints on the output values. As one might expect from the last section, these constraints are equality, membership, and subset, and any occurrence of one of these operators is immediately evaluated into a constraint, with the argument that contains no references to post-state values evaluated into a value of one of the abstract types. Note that this transformation is only applied to constructive expressions as defined in the previous section, so an appropriate argument with no references to post-state values is guaranteed to exist. For the same reason, the transformation need deal only with `/\` and quantified assertions, and an `/\` assertion is handled by simply appending the lists of constraints generated by its two arguments. For example, given the the post-condition `header(s') = <1, 2> /\ last(s') = 3`, the transformation produces the constraint list `[header(s') = <1, 2>, last(s') = 3]`.

Universally quantified assertions are transformed to constraints by repeatedly evaluating the body of the assertion with the variable bound by the quantifier bound to each value in the domain, in turn. The resulting lists of constraints are appended into a single list of constraints. So, if (in the post-condition of `RelTo`) the default

parameter to RelTo is {(1, 2), (2, 2), (2, 3)} and the value of key is 2, then the assertion

```
p.First() = key => p.Second() \in result'
```

is evaluated 3 times, with p bound to (1, 2), (2, 2), and (2, 3) in turn. The resulting constraint list is [2 \in result', 3 \in result'].

For existentially quantified assertions, the technique is to repeatedly evaluate the body of the assertion for its boolean value, where subexpressions involving post-state values are ignored. This is done with the quantified variable bound to each element of the domain, in turn, until the evaluation returns true. Then, the element that satisfies the body is bound to the quantified variable, and the body is evaluated to generate constraints. So, for example, an assertion such as:

```
\exists (int x) [(1 <= x <= 3) /\ x mod 2 = 0 /\ y' = x + 5 ]
```

causes the predicate x mod 2 = 0 to be evaluated for x equal to 1, 2, and 3. As only 2 satisfies this predicate, the constraint list [y' = 7] is generated. If multiple elements of the domain satisfy the predicate, the first one that this evaluation technique finds is used. Note that such an existentially quantified assertion constitutes an underdetermined or non-deterministic specification, and that this is the only kind of "loose" specification that the execution technique can use constructively in its current form. With the addition of backtracking, specifications that are loose because of assertions using \/ and => as described earlier can also be used constructively. The addition of backtracking will also allow more sophisticated use of existentially quantified assertions. Many elements of the domain quantified over can potentially satisfy the parts of the body of the assertion that do not refer to post-state values. In case the original choice of such an element causes the execution of the post-condition to fail, backtracking would allow other elements of the domain to be tried.

4.3.2.3 Evaluating Constraints into Values The execution algorithm next tries to construct a post-state value for identifiers appearing in the modifies clause and for any non-void function result, using the list of constraints constructed as described in the previous section. The first step is to check whether the (primed)

identifier appears directly in one or more constraints. If so, then there are only two possibilities, assuming that the post-condition used to generate the constraints is satisfiable. Either one or more identical = constraints match, and the (identical) post-state value is directly in each such constraint, or a group of `\in` and `\subset` constraints match, and so a value of type set needs to be constructed. This is done by unioning together the first arguments of all the constraints that matched. For example, if the list of constraints is `[2 \in result', 3 \in result']`, then the post-state value constructed for `result'` is `{2, 3}`.

If the primed identifier doesn't match any constraint and the post-state value under construction is of type tuple or sequence, then the next step is to check the constraint list for applications of built-in tuple or sequence observers, respectively. (Recall that these observers are defined in Section 4.2.) The only non-atomic values in SPECS-C++ are tuples, sequences, and sets, and the only built-in observers of sets (`\in` and `\subset`) are converted directly into constraints, so tuples and sequences are the only types whose values can be constructed in this way.

The only built-in way to observe tuples is to look at their components, or fields. Thus, the next step in trying to construct a post-state value of a tuple type is to attempt to match (in the constraints list) the application `field(identifier')` for each field of that type of tuple. If this is successful, then the post-state tuple value can easily be constructed.

When trying to construct a post-state value of a sequence type, our algorithm constructs applications of the sequence observers to the appropriate primed identifier, and then matches these applications with the constraint list. Each successful match defines a part of the post-state value. For example, if the primed variable of interest is `s'` and the constraint list is `[header(s') = <1, 2>, last(s') = 3]`, then two applications match, and the post-state value of `<1, 2, 3>` can easily be constructed. Clearly, such multiple matches can produce redundant information about the post-state value of a sequence, and this information should be checked for consistency. This has not yet been implemented, but seems straightforward. The other case (some part of the post-state value of a sequence is not defined in the constraint list) is treated next.

If some part of the post-state value of a sequence or tuple is completely uncon-

strained by the constraint list, then there are two possibilities, depending on whether the associated identifier (formal parameter of a tuple or sequence type) is defined (has a value associated with it) in the pre-state. If the identifier is defined in the pre-state, then the "and nothing else changes" semantics of SPECS-C++ implies that the unspecified part should be whatever it was in the pre-state value. If the value is being constructed from scratch, then the post-condition didn't provide enough information to construct it, and so the construction fails and reports an error.

Post-state values of union types are constructed by searching the types of the union, and finding the first one to which the value belongs. If none of the type works, then the attempt to construct the post-state value fails. This is another situation in which backtracking could be used to make more specifications executable.

The steps described in the previous paragraphs are applied in a mutually recursive fashion. Thus, if we had modeled integer relations as a sequences of sets of integers (i.e. each index position in the sequence is related to each of the elements of the set found at that index position), then our algorithm could utilize constraints like `3 \in index(R', key)` in constructing a post-state value for `R'`.

If the post-state value of some identifier appearing in the `modifies` clause is completely unspecified (i.e. that identifier does not appear in any constraint), then it is handled in the same way as unspecified parts of tuples or sequences – if it is defined in the pre-state, then the post-state value is the same as that of the pre-state value, and if it is not defined in the pre-state, then the construction fails. Note that the `result'` of the member function being specified is always undefined in the pre-state, and so member functions with non-void return types must always construct a post-state value for `result'`.

4.3.2.4 Checking the Nonconstructive Parts Next, the parts of the post-condition that were determined to be nonconstructive are checked by evaluating them for their boolean values. This evaluation is done with respect to *both* the pre- and post-states, as the truth value of the nonconstructive parts may depend on both. Unprimed identifiers have their pre-state values, and primed identifiers (including `result'`) have their post-state values. So, for example, the nonconstructive portion of the post-condition for `RelTo`:

```
\forall (int i) [
  (i \in result') =>
    \exists (OrderedPair p) [
      (p \in theRel) /\ i = p.Second() /\ key = p.First()]]
```

is evaluated w.r.t a pre-state where `theRel` is `{(1, 2), (2, 2), (2, 3)}`, `key` is 2, and a post-state where `result'` is `{2, 3}`. The evaluation mechanism is identical to that used for pre-conditions. If this evaluation returns true (as it does for this example), then all is well. Otherwise, the attempt to execute the post-condition fails. This can occur because the post-condition isn't satisfiable, or because the technique described for finding post-state values failed to find satisfactory ones.

4.3.2.5 Constructing the Result State To complete the execution of a member function call, we need to construct the state that results from the call. This is different from the post-state we have been discussing so far, as this is outside the scope of the member function. Thus, the formal parameters are no longer part of the state, and any side effects of the member function are now reflected in the actual parameters, rather than the formal parameters. Note that any formal parameter that is mutated must appear in the `modifies` clause of the called function. If the member function that was called has a non-void return type, then the value constructed for `result'` is just printed, as return values of functions do not affect the state.

As all state modifications performed by the member function have been made and any return value constructed, execution of the member function call is complete.

4.4 Limitations of the Execution Technique

While the work described here makes a large subset of SPECS-C++ executable, not all valid SPECS-C++ specifications can be executed. A post-condition must contain constructive subassertions that define all of the post-state values to be found. The following list of nonconstructive (and so, non-executable) assertions is provided to highlight the limitations of the execution technique.

- Quantified assertions where the domain of the bound variable depends on post-state values, implications where the antecedent refers to post-state values, and

multiple occurrences of primed identifiers in the same constraint (in the sense of Section 4.3.2.3). Examples include:

```
\forall (int x) [(x \in (foo' \union {3})) => x \in foo2' ]

x' = 3 => y' = 4

index(S', i') = 3
```

Augmenting the execution technique with backtracking would only partially solve these problems. Many more such assertions would be executable by a more incremental approach: find post-state values for all the identifiers possible, and then try again in a state where these primed identifiers are bound to the value found. This corresponds to finding all of the post-state values that depend only on pre-state values, then finding all the post-state values that depend only on pre-state values and those post-state values found in the previous step, and so on until no more post-state values can be found. Note that this scheme is guaranteed to terminate, as at least one post-state value must be found at each step for this iteration to continue.

- Negated assertions and assertions in which any relational operator other than =, `\in`, or `\subset` is used to describe the relationship of some value to the post-state value being constructed. Examples of such assertions can be found in Section 4.3.2.1. They limit the possible post-state values that can be constructed. As these limits are checked when the nonconstructive portion of the post-condition is evaluated, the evaluation technique already utilizes such assertions as well as can be expected in the absence of constraint-satisfaction techniques such as those discussed in [Lel88].

- Post-state values used as arguments to abstract function references and member functions calls in post-conditions. Assertions like `someabsfun(foo')` do not help in constructing the post-state value for `foo`, at least as far as the work described here is concerned. Note that simply replacing the call or reference by the specification of the associated function will not work. To see this, consider

a post-condition of some member function of class `Relation` that just mimics `RelTo` on a `key` of 2, i.e. `result' = self.RelTo(2)`. The post-condition of `RelTo` is a (boolean valued) assertion, so this new post-condition fails to type check. The same problem arises with abstract function references, as an abstract functions can also define its return value in the implicit manner typified by `RelTo`. However, this problem does seem reasonably tractable, and so this is an area for further research.

4.5 Related Work

The most closely related work is the **fase3/C++** language of Kamin and Kraus [KK93, Kra88], as it is a (mostly) executable interface specification language for C++ classes. The **fase3** approach is very similar to that taken in the Larch [GHW85, GHG+93] family of specification languages, as it is two-tiered, consisting of the **fase3** shared language and the bf fase3/C++ interface language for C++. The shared language is where most specification occurs, and so where interesting execution occurs as well. Only a few primitive types such as integer, boolean, and so on are built into **fase3**. More structured types, such as the set, sequence, and tuple types of SPECS-C++ are specified in the shared language using a unique style of algebraic specification that allows any type to be represented as a tuple of functions. As long as these functions remain finite for a particular element of the type, the functions can be represented as "tables" (finite sets of tuples), and quantified assertions over that element can be executed. The syntax for quantified assertions is elegant and concise, as the specifier need not supply an explicit bound on the quantified variable, as is required in executable SPECS-C++ specifications. Assertions that use observers to define values are also executable — in fact, the functions that represent an element are exactly the observers of the type as applied to that element.

However, many quantified assertions that are executable using our technique for SPECS-C++ can not be executed in **fase3**. Given the natural **fase3** shared language specifications of set, sequence, and tuple, the kinds of quantified assertions that have no references to post-state[7] values that are executable seem quite similar to those

[7] In **fase3**, there is no concept of pre- and post-state at the shared level. When

that are executable using our technique for SPECS-C++. However, only one kind of **fase3** quantified assertion can be evaluated if it contains references to post-state values. This kind of assertion is a restricted form of existential quantification that can only be used to select a particular element from the domain quantified over. Thus, any other kind of quantified assertion can only be evaluated for its boolean value, and so does not provide any interesting way to constrain post-state values. For example, the post-condition of the member function `RelTo` as discussed in the previous section, is not executable using the **fase3** execution technique.

The execution techniques are quite different. In **fase3**, specifications are first compiled to an extended λ-calculus, and then to an extended form of combinator graph, which is then reduced. An execution technique of this generality seems to be necessary because the specifier defines the observers of a particular type, rather than the observers being built into the language as in SPECS-C++. Because of the compilation phases and complicated reduction phase, the **fase3** execution technique seems unlikely to execute specifications as rapidly as our technique for SPECS-C++. We are also unsure of the usefulness of errors reported by the reduction algorithm in debugging the specification that the graph was derived from.

Another work with some relation to our own is the structural mapping from Object-Z [CDD+90] to C++ of Rafsanjani et. al. [RC93]. This mapping is an informal guideline for producing C++ implementations from Object-Z specifications. Object-Z classes are mapped to C++ classes, operations to virtual member functions, and so on. The mapping is not intended to be an automated translation, so the only tool support provided is a partial implementation of the Object-Z basic types. The work does provide some interesting observations on the relationship between specification inheritance and code inheritance which may be helpful as we work on adding inheritance to our interpreter for SPECS-C++.

---

we refer to post-state values in the context of **fase3**, we are referring to the values returned from shared language functions. These are the values that are defined by specifications, and so play the role of post-state values in SPECS-C++.

4.6 Conclusion

One of the strengths of the execution technique described here is that it is relatively efficient (at least with respect to Prolog) and execution times are predictable. The most expensive built-in operators (set intersection, difference, and subset) take $O(n^2)$ time, execution of quantifiers takes time linear in the size of the domain quantified over, and all other built in operations take either linear time (length of sequences, size of sets, etc.) or constant time. Hence, running times can be estimated by inspection of the specification. With the addition of backtracking to the interpreter, some of this predictability would be lost. However, the loss of efficiency would be minimal, as any post-condition that is currently executable would not require backtracking for correct execution by the augmented interpreter.

However, this execution technique does lose some aspects of expressiveness as compared to Prolog, even with the addition of backtracking. For example, a classic way of implementing sorting in Prolog is to define predicates "permutation" and "ordered", and then define a sorted sequence as an ordered permutation of the original. SPECS-C++ is expressive enough to state the specification of a sort this way – the post-condition would look like:

```
Permutation(s, s') /\ Ordered(s')
```

for a sequence argument `s` and appropriate abstract functions `Permutation` and `Ordered`. This specification is not executable by the technique described here. On the other hand, executing the Prolog sorting program takes exponential time, and so there is a trade-off between expressiveness and execution time.

Additionally, our execution technique was designed to take advantage of the ways in which humans usually write specifications. We have found, for example, that people tend to write more constructive specifications (in the sense developed in Section 4.3.2.1), rather than nonconstructive specifications like the one discussed in the previous paragraph. In particular, our technique was developed in the context of a suite of specification examples used for teaching formal methods at both the undergraduate and graduate levels. Almost all of these specifications are executable with only (very) minor modifications, such as placing parenthesis around the domains

of variables bound by quantifiers. This provides good evidence that the executable subset of SPECS-C++ is useful in practice.

We have provided an interpreter for SPECS-C++, rather than using translation to some programming language. Hence, this technique does not require the user to know Standard ML (the language the interpreter is written in), and also can more easily report helpful error messages than techniques using translation.

The execution technique also demonstrates the use of default frame axioms in an executable specification language. Frame axioms are used to say "and nothing else changes" – that only the state transformations explicitly required by the post-condition actually occur, and no more. The modifies clause is a form of frame axiom, as it explicitly limits the side effects of a member function to only the formals (and corresponding actuals) that occur in the modifies clause. However, additional frame axioms are often required, as member function specifications don't always completely specify all post-state values. While extra explicit frame axioms can be included in the post-condition, doing so leads to large and unreadable specifications, along with a number of other problems [BMR93]. In SPECS-C++, implicit frame axioms are included in the semantics of the specification language. These frame axioms are also part of the executable subset of SPECS-C++. This is demonstrated in the section on evaluating constraints into post-state values (Section 4.3.2.3). For sets, the default frame axiom is to find a minimal set satisfying the post-condition. To see this, note that nothing is included in a post-state set value unless the post-condition explicitly includes it, in the form of a `\in` or `\subset` expression. For tuples and sequences, the default frame axiom is that no field or index position changes from the pre-state to the post-state unless the post-condition explicitly specifies that it change.

Two major extensions of the interpreter are planned. These extensions will add constructs for dealing with SPECS-C++ abstractions of C++ objects and for specification inheritance.

In C++, "An *object* is a region of storage" [Str91]. Equivalently, an object may be thought of as a cell in memory, or a value and its address. In SPECS-C++, we are concerned with distinguishing between objects and values because objects can be mutated and aliased, while values cannot. Incorrect usage of mutation and aliasing is a common cause of errors in C++ programs, so it is important to advertise to

users of a class (through the specification of the class) exactly where mutation and aliasing can occur. SPECS-C++ borrows the C++ notion of references as a uniform mechanism for creating and handling objects. However, references are not yet part of the executable subset of SPECS-C++.

Just as in C++, in SPECS-C++ inheritance relationships are established through the use of derived classes. The syntax and semantics used are similar to C++, so that a class specified as a derived class inherits the abstract data members and (the specifications of the) member functions of its base (super) class [Lea93]. We plan to add specification inheritance to the SPECS-C++ interpreter, and so to the executable subset of SPECS-C++.

Even in its unfinished state, the SPECS-C++ interpreter can already be used in two important ways. The first is that it provides a formal semantics for a large subset of SPECS-C++. Even if the specifier never wants to execute specifications, this work is useful in that it gives denotational and operational semantics for the executable subset of SPECS-C++, which includes the built-in operators, abstract functions, using member functions in specifications, and (soon) objects and inheritance of specifications. As the SML compiler provides operational semantics for all the constructs in the executable subset, this work has a decided advantage over non-executable descriptions in reference manuals. The SPECS-C++ interpreter is an unambiguous specification of the meaning of the executable subset of SPECS-C++ specifications.

Secondly, this work provides an executable subset of SPECS-C++, and the means to execute specifications written in the subset. Thus, a specification can serve as a prototype of the finished system. Some of the advantages for the specifier and client are [WBL93]:

1. Validating specifications. The specifier can now test and debug a specification in much the same way that a programmer would validate a program. This pushes validation into the specification stage of the software development process.

2. Understanding formal specifications. The client, who is likely to have little or no experience with formal methods, now has a way to understand a formal specification. By experimenting with the prototype, the client can discover

and report to the specifier erroneous or unexpected results, and missing or incomplete features.

Both these points imply that requirements and specification errors are likely to be discovered earlier, resulting in less expensive and more reliable software. Additionally, problems that traditional software engineering techniques cope with poorly, such as missing functionality and incomplete requirements documentation, are much more likely to be addressed by an executable specification. Often, the software developer is completely unaware of such problems, and only discovers them after the completed software is delivered to the client. Giving the client a prototype can result in such problems being discovered far earlier.

Thus, this research is a contribution to both the theoretical and practical sides of formal methods. On the theoretical side, it demonstrates an executable specification language and execution technique with clear advantages over other approaches. On the practical side, it demonstrates that prototypes generated by executable specifications can be efficient enough for practical use. As such, it represents solid progress in applying formal methods to industrial software production.

5. APPLICATIONS OF CONTRAINT LOGIC PROGRAMMING TECHNIQUES IN EXECUTING FORMAL SPECIFICATIONS

A paper in preparation for ACM Transactions on Software Engineering and Methodology

Tim Wahls,[1] Gary T. Leavens, and Albert L. Baker

Abstract

Executable specifications may be a key to the industrial acceptance of formal specification techniques, as executability leads to better and more useful of specifications. However, executability should not compromise the high level of abstraction typically achieved with model-based specification languages. We investigate the application of techniques from constraint logic programming to the execution of specifications, and show how variants of these techniques can be used to execute very high level specifications.

5.1 Introduction

Despite the strong research interest in the use of formal specification languages in developing software, such languages have made relatively little impact in industrial development settings. The advantages of formal specifications are clear: precise and unambiguous descriptions of software functionality. However, from an industrial viewpoint, specification activities produce little that can be demonstrated to clients as progress toward a finished product, and training software developers in the formal notations involved is an expensive overhead. Additionally, formal specifications are

[1] Wahls's work is supported by a fellowship provided by IBM Rochester.

written by humans, and so suffer from "bugs" analogous to those that infest programs. An erroneous specification, no matter how precise, is of little value.

How can such problems be overcome? A partial answer, we believe, lies in executable specifications. An executable specification serves as a functional prototype of the final system, which allows client and developer interaction on system functionality in the earliest stages of development. Currently, this kind of interaction occurs largely after at least partial implementation, often resulting in wasted programming effort. Executable specifications can serve as test oracles, which allow partial automation of the process of testing implementations [GB94]. Executable specifications can also be debugged in much the same way that programs are debugged, resulting in the early detection and correction of errors. This is vital in software development, as the cost of correcting an error in maintenance is typically 100 times greater than the cost of correcting the corresponding error at the specification level [Boe81].

It is natural to think of the post-conditions of model-based formal specifications as constraints to be satisfied by the post-state. Thus, the execution of such specifications corresponds to a constraint satisfaction problem. However, much of the work in logic and Constraint Logic Programming (CLP) languages is not directly applicable, as the domains involved are different. For example, a common domain in CLP is the rationals or reals, and the constraints to be satisfied are systems of equations and inequalities. While such situations do arise in specifications and can be described in languages such as VDM and Z, specifications over domains such as sets, sequences, tuples, and objects are far more common in industrial software development.

There is also a basic difference in the kind of answers that are desired between the areas of constraint satisfaction and executable specifications. For example, specifications are usually intended to be implemented in an imperative programming language, and so define mappings from procedure inputs to outputs. Thus, the ability to invert logic programs (or specifications) – i.e., to produce the inputs corresponding to given outputs – is of less interest in executing specifications than it is in logic programming. Additionally, as specifications define the functionality of imperative procedures, the user of an executable specification language usually wants a particular post-state value as an answer, rather than constraints on suitable post-state values.

However, some of the techniques used in CLP languages can be of great value

in executing specifications. The rest of this paper describes the application of CLP-like techniques in executing formal, model-based specifications. In Section 5.2 we provide a brief introduction to Prolog and modern CLP languages such as CLP($\mathcal{R}$) and Prolog III. In Section 5.3 we discuss model-based specification languages and the features of such languages that complicate the application of CLP techniques. In Section 5.4, we describe our technique for executing specifications. Section 5.4.3 provides some example SPECS-C++ specifications and describes their execution. Section 5.4.5 characterizes the run time performance of the algorithm. Section 5.4.6 describes some of the current limits of the execution technique, and Section 5.4.7 outlines future work on lifting these limitations. In Section 5.5 we conclude with a comparison of our technique and logic and constraint logic programming techniques.

5.2 Prolog and CLP Languages

The best known logic programming language is Prolog [CM84, Col85, Coh85]. Prolog is based on Horn clauses, which represent a subset of first order predicate logic. A Prolog program consists of a finite collection of rules. A *query* is a list of atomic formulas, and an answer to a query is a substitution of values for the free variables of the query that allow it to be concluded from the rules in the program. The domain of computation is finite trees, where trees are simply terms.

The basic execution mechanisms of Prolog are unification, refutation (also known as resolution), and backtracking. Unification is the process of finding a substitution for the free variables of two atomic formulas or terms that make the atomic formulas or terms identical. Refutation is the deductive process by which conclusions are drawn from programs. The refutation algorithm often has a choice of rules to apply, so that when a choice of one rule results in the inability to conclude the query, the algorithm can backtrack and try a different rule.

The best known CLP languages are Prolog III [Col90] and CLP($\mathcal{R}$) [JMSY92]. While the languages have some similarities — for example, both use refutation and backtracking much like Prolog — they are also quite different, both from each other and from Prolog.

In Prolog III, unification is completely replaced by constraint solving. A con-

straint is any atomic formula (including arithmetic equations and inequalities) . Rules and queries can include an explicit set of constraints. The answer to a query is a set of constraints on possible answers (in a solved form). As the execution of a Prolog III program progresses, backtracking occurs whenever the current set of constraints becomes unsatisfiable, or when no rule can be applied. The evaluation of a constraint can be explicitly delayed by the user with a "freeze" mechanism [Coh90]. The domains over which constraint solving occurs are infinite trees, lists, booleans, reals, characters, and strings.

On the other hand, CLP($\mathcal{R}$) uses constraint satisfaction only for the reals, and unification for other computational domains. Constraints consist of arithmetic equations and inequalities, and equality of nonarithmetic terms. Constraints can be used anywhere in the body of a rule. As in Prolog III, answers to queries are simplified sets of constraints. Constraints are not explicitly delayed, but non-linear equations are delayed until sufficient variables are solved to make them linear. The domains of computation are the reals and finite trees (as in Prolog).

With respect to CLP languages, part of our contribution is the formalization of constraints over specification language domains such as sets, sequences, tuples, and objects, and the introduction of techniques for solving these constraints.

5.3 Model-based Specification Languages

Model-based specification languages describe the functionality of procedures in terms of an underlying mathematical model. The best known examples of these languages are VDM [Jon90] and Z [Hay87, Spi89, Spi92], although many other model-based specification languages have been developed. The model typically includes primitive types such as integers and characters, as well as more structured types such as finite sets, sequences, and tuples. The constants of and functions and relations on the model types provide the vocabulary for writing first order predicate logic pre- and post-conditions. These conditions describe, respectively, sufficient conditions for the specified procedure to execute correctly, and what is guaranteed to be true when the procedure terminates correctly. In other words, the pre-condition describes what must be true of the pre-state (the program state just before the specified procedure

is called) for the procedure to execute correctly, and the post-condition describes the post-state resulting from the execution of the procedure. Hence, execution of such specifications corresponds to finding post-state values that satisfy the post-condition.

The execution technique described in the next section was developed in the context of the specification language SPECS-C++ [WBL94], which is a formal and model-based language specialized for specifying the interfaces of C++ classes [Str91]. However, the execution technique is equally applicable to other model-based specification languages, as it depends only on first order assertions over a fixed set of model types. In fact, the addition of C++ specific features only complicates the execution of specifications. These features include classes, objects, and inheritance. Of these, only objects are discussed in this paper.

Object types are used in SPECS-C++ to model aliasing and mutation at the specification level. In this context, an object is a container for a value. Objects are the same as locations in denotational semantics [Sch86]. Mutation occurs when the value held by an object changes from the pre-state to the post-state. Aliasing means that an object can have multiple names, such as an element of a sequence and a field of a tuple. That is, the expressions `index(S, 2)` and `count(T)`, where `S` is a sequence of objects and `T` is of a tuple type that has a field `count` of an object type, can denote the same object. If the value of this object is changed (mutated), evaluating either of these expressions will return the new value. Specification languages such as VDM and Z allow the specifier to distinguish between pre-state and post-state values, but do not have a complete and uniform theory of objects, and do not allow the detection or specification of aliasing. However, aliasing is characteristic of C++ programs, and the ability to fully specify aliasing is vital in writing practical specifications of C++ classes.

Thus, objects (and constraints on objects) are a complication in executing SPECS-C++ that is not found in Prolog III or CLP($\mathcal{R}$). Other domains in SPECS-C++ that are not typically found in CLP languages include integers, sets, sequences, and tuples. In the case of integers, the cause of this omission is most likely that the truth value of first order predicate logic assertions over integers is undecideable [Kle52]. However, as integers are a domain of SPECS-C++ (and VDM and Z), we want to handle a subset of assertions that explicitly involve integer variables in executing

specifications. The other domains of SPECS-C++ can be implemented by trees, and this is likely why they have not been included as separate domains in CLP languages.

As CLP languages are typically untyped, they can not use type information as such in solving constraints. However, this information can be useful in constructing values that satisfy constraints. For example, knowing that a value has a particular tuple type gives both the number and type of the fields of the tuple. In languages such as SPECS-C++ that use union (alternative) types, type information also allows sensible backtracking when type errors occur.

Another important difference between SPECS-C++ specifications and CLP programs is that CLP programs consist of rules with a simple structure, while SPECS-C++ specifications use first-order predicate logic syntax (including explicit quantifiers). In the case of Prolog III, the constraints are even explicitly separated from the rest of the program. From an execution standpoint, this means that any attempt to apply CLP techniques to executing specifications must start with some way of extracting constraints from specifications.

5.4 Executing Model-Based Specifications

The execution of a SPECS-C++ procedure specification is simply the execution of a call to that procedure. Executing the call has two steps.

1. Check that the pre-state and actual arguments satisfy the pre-condition.

2. Construct post-state values that satisfy the post-condition.

Evaluating the pre-condition is relatively easy, as all the values involved are known, and in the executable subset of SPECS-C++, all quantified variables must range over a finite (and explicit) domain. Thus, the pre-condition is simply evaluated for its boolean value. On the other hand, the execution of post-conditions is much more complicated, and it is here that CLP-like techniques are helpful.

A summary of the algorithm for satisfying post-condition assertions is summarized in Figure 5.1. The algorithm is based on iteratively transforming parts of the post-condition into constraints, and then solving those constraints into post-state values. As more (parts of) post-state values become known, parts of the post-condition

- while part of the post-condition remains do:
  - generate constraints:
    - transform some parts of the post-condition into constraints, and remove these parts from future consideration
    - directly evaluate some parts of the post-condition, and remove these parts from future consideration
    - delay any remaining parts of the post-condition until future iterations
  - solve constraints:
    - solve any constraints that are currently solvable
    - delay other constraints until future iterations
- solve any remaining constraints
- apply implicit frame axioms (SPECS-C++ only)
- return the post-state values found

Figure 5.1: The algorithm for satisfying post-conditions.

that refer to those values can be evaluated or transformed into constraints. After all of the post-condition has been transformed into constraints or directly evaluated, the post-state values found are returned. The algorithm is explained in more detail in the following discussion.

5.4.1 Conversion of the Post-condition to Constraints

The first step in understanding the execution of post-conditions is understanding the observers[2] of the built in model types, as these observers form the basis of constraints. The basic observer of all types is equality (=). The only additional observer of tuples is the `field` function – i.e. the extraction of a given field from the tuple. For objects, the observers are the postfix functions ^ and ', which, respectively, extract the pre- and post-state values of an object. The observers of sets are `member`, `subset`, and `size` (cardinality). Besides the natural `length`, `member`, and (1-based) `index` functions on sequences, SPECS-C++ defines additional sequence functions as follows, where <> is sequence constructor notation, and || is used for appending sequences:

```
first(<a>||s) = a
last(s||<c>) = c
header(s||<c>) = s
trailer(<a>||s) = s
```

Given these definitions, each constraint has one of the following forms:

val `member` *lhs*
val `subset` *lhs*
lhs = *val*
`length(`*lhs*`)` = *val*
`size(`*lhs*`)` = *val*

where *val* is a total value. Total values will be defined precisely in Section 5.4.2, but roughly they are values that are completely known. An *lhs* is an expression denoting a post-state value or part of a post-state value, and has one of the following forms:

[2]Observers are functions that, when applied to a value of the appropriate type, allow one to "look at" some aspect of the value.

ident
lhs'
`field(`*field-name, lhs*`)`
`index(`*lhs, val*`)`
`first(`*lhs*`)`
`last(`*lhs*`)`
`header(`*lhs*`)`
`trailer(`*lhs*`)`

where *ident* denotes a variable name, and *field-name* is just the name of a field of some tuple type. Constraints are further restricted by typing (as not every expression of the form given is correctly typed), and by the exclusion of **member** constraints whose *lhs* is of type sequence. Note that the only arithmetic constraints are of the form `x = `*val*, where *val* can be directly evaluated into a number. Thus, the execution technique does minimal numerical constraint solving, and this is an area where existing CLP techniques could be used to advantage.

To simplify finding post-state values, the post-condition is transformed into a set of constraints. These constraints can then be solved without reference to the structure of the post-condition. To ensure that this process produces post-state values that satisfy the post-condition, this transformation must have the property that if the set of post-state values produced satisfy the constraints, then these post-state values satisfy the post-condition. To show how this property is achieved, we develop the constraint generation algorithm by induction on the structure of post-condition assertions. The basis of this induction is atomic formulas (in the sense of logic programming), which are handled in one of three ways.

- The atomic formula can be converted to a constraint. For this to occur, the atomic formula must have the form of a constraint, except that a term appears where *val* appears in the grammar for constraints. This term can only refer to pre-state values and the known parts of post-state values, so that it can be directly evaluated into a total value. The constraint is generated and this atomic formula need not be processed further.

```
void abs(int& x);
pre: true
modifies: x
post: (x^ >= 0 /\ x' = x^) \/ (x^ < 0 /\ x' = -x^)
```

Figure 5.2: The specification of an absolute value procedure. Type `int&` is the type of objects containing integers.

- The truth value of the atomic formula can be determined from the values that are currently known. If the atomic formula is true, it need not be processed further. If it is false, backtracking occurs.
- Otherwise, processing of the atomic formula is delayed until after the current set of constraints has been solved into post-state values. Thus, processing of the post-condition is an iterative procedure, and continues until either the entire post-condition has been used up or until no new constraints can be generated.

The induction steps of constraint generation handle the boolean connectives and universal and existential quantifiers. For example, conjuncts can be considered separately as all must hold for the post-condition to hold. Each disjunct is first checked to see if it is false using a nonstrict evaluation procedure. For example, in the natural specification of an absolute value procedure of Figure 5.2, any pre-state value of x makes one disjunct of the post-condition false, and this disjunct can simply be discarded. (Recall that ^ and ' are postfix functions for extracting the pre- and post-state values of an object, respectively.) If multiple disjuncts are not false, then one is chosen and a backtrack point is created so that the execution procedure can return and try one of the remaining disjuncts if the current choice fails.

Logical implications are handled similarly, except that the execution procedure tries to determine the truth or falsity of the antecedent first. Specifiers rarely write logical implications whose antecedents refer to post-state values, so this attempt is

usually successful, and allows easy and efficient processing of implications. If the antecedent can be determined to be true, then it can be discarded and the consequent processed recursively. On the other hand, if the antecedent is false, then the entire implication can be discarded. If no determination can be made, then both the antecedent and consequent are assumed to be true, and a backtrack point is created so that the execution procedure can return and try the assumption that the antecedent is false.

Negated assertions are never transformed into constraints. Instead, they are delayed until sufficient post-state values are known to allow them to be evaluated directly. When this occurs, if the negated assertion is satisfied, then it is discarded and constraint generation proceeds. If it is not satisfied, backtracking occurs. If any delayed assertions remain and no more constraints can be generated (so this holds for atomic formulas as well), then the execution algorithm reports that it can not execute the post-condition.

Quantified assertions can only be evaluated if the domain that the bound variable ranges over is known and finite[3], and if the body of the quantified assertion can be evaluated either for its boolean value or into constraints. Thus, evaluation of quantified assertions is delayed until this occurs. When a universally quantified assertion is evaluated or transformed into constraints, its body is evaluated once for each element of the domain, with the quantifier variable bound to each element in turn, so that every element of the domain is used in generating constraints. For existentially quantified assertions, the execution technique finds some element of the domain, that, when used as the value of the bound variable, does not make the body of the assertion false. This is checked using the procedure that is used for disjuncts. If multiple elements of the domain satisfy this property, then a backtrack point is created so that each satisfying element can be tried if necessary. Then, constraints are generated from the body of the assertion with the quantifier variable bound to

[3]Thus, the bound variable is somewhat like a domain variable in the CLP language CHIP [Van89] in that it ranges over a known and finite domain. However, our use bound variables is quite different from the use of domain variables in CHIP. Domain variables are still logical variables, and thus are used in unification rather than in generating constraints.

```
void sort(seqint& si);
  pre: noDups(si^) // noDups returns true if sequence has no duplicates
  modifies: si
  post: \forall int x [ x member si^ =>
          \exists int i [1 <= i <= length(si^) /\
                         index(si', i) = x]]
     /\ \forall int i [2 <= i <= length(si') =>
           index(si', i - 1) < index(si', i)]
     /\ length(si') = length(si^)
```

Figure 5.3: The specification of a sort procedure. Here, `seqint` is the type of sequences of integers.

the chosen element of the domain.

For example, consider the specification of a procedure for sorting sequences of Figure 5.3. The first and third conjuncts can immediately be converted to constraints, but the second can only be evaluated after a post-state value for `si` has been found. Note that there are many possible constraint sets that can be produced, and so a great deal of backtracking can occur. For example, if the pre-state value of `si` is `<3, 2, 1>`, then one possible constraint set is:

`{index(si', 1) = 3, index(si', 1) = 2, index(si', 1) = 1, length(si') = 3}`

which clearly has no solution. Another possibility is:

`{index(si', 1) = 3, index(si', 2) = 2, index(si', 3) = 1, length(si') = 3}`

which has a solution, but that solution does not satisfy the second conjunct of the post-condition. In either case, backtracking will occur.

5.4.2 Solving Constraints into Post-state Values

After a set of constraints has been generated from the post-condition, they are solved into partial post-state values. These values are partial because many constraints yield only part of a post-state value. For example, the constraint:
`index(s', 3) = 2`
defines only one index position in the sequence `s'` and does not constrain the total length of the sequence (other than to set a lower bound of three). Thus, a partial sequence value is a prefix of the total value which can also contain partial values. Thus, partial sequences are similar to improper lists in Prolog III. Partial tuple values are simply tuples with one or more fields containing partial values. Similarly, partial object values are objects containing partial values. Finally, partial set values are simply subsets of the total value. Sets are not allowed to contain partial values because there is no way to refer to a set element that is independent of the value of that element. An element of a sequence can always be retrieved if one knows nothing more about it than its index position, but there is no analogous way to retrieve set elements. Thus, if a partial value is contained in a set, there is no way to use constraints to further define that element.

As the execution procedure is iterative, on each iteration after the first the procedure may need to evaluate expressions involving partial values. Let `undef` denote the completely undefined (unconstrainted) value. Equality checking of partial values uses the following rule: for any value `v` (including `undef`), the value of `v = undef` is `undef`. The evaluation of tuple and object observers is straightforward, as they are allowed to return partial values, and return `undef` when applied to `undef`. For example, let `("first" 2, "second" undef)` be a tuple with fields `"first"` and `"second"` with the indicated values for each field. Then, the value of `field("first", ("first" 2, "second" undef))` is 2, and the value of `field("second", ("first" 2, "second" undef))` is `undef`.

For partial sets and sequences, the situation is more complex. Let $S_{\texttt{undef}}$ denote a set or sequence of unknown `size` or `length`, and S_n denote a set or sequence of `size` or `length` n. For example, $\{1, 3, 5\}_{10}$ is a set of `size` 10 containing {1, 3, 5} as a subset. Given this notation, the operations on partial set values are summarized

in Table 5.1, and the operations on partial sequences in Table 5.2. In the table, the result of dropping the subscript of a partial set value is the defined subset of the partial set considered as a total set value. The result of dropping the subscript of a partial sequence value is the defined prefix of the total sequence value, which can of course still contain partial values. To distinguish these prefix values from total values, the names of SPECS-C++ sequence operators will be italicized when they are applied to prefix values, and *app* will be used to denote appending of these values. Cases where one or more arguments are `undef` are omitted, as the result is always `undef`. In particular, the `size` (`length`) of a partial set (sequence) value is `undef` unless it has been defined by a previous `size` (`length`) constraint, as described later in this section. Note that partial values also complicate set difference (denoted by -), as care must be taken to ensure that any partial set value returned is in fact a subset of any possible answer for all possible total-valued arguments. Thus, if the second argument is partial, the result is always `undef`.

Thus, the problem of constraint solving in this context comes down to using a constraint to further define a partial value. The *lhs* part of the constraint is used to find the appropriate partial value by name, and part of the *lhs* may be used to navigate within that partial value find the part of the value that the constraint constrains. The rest of the observers appearing in the *lhs* are "used backwards" to help in further defining a partial value. For example, given the constraint `field("foo", first(result)) = 3, sequence(tuple(int "foo", float "bar"))` as the type of `result`, and a previous partial value of:

`<undef, ("foo" 7, "bar" 6.3)>undef`

for `result`, `first` is applied to this partial value, yielding `undef`. Next, the inverse of the `field` function is used to solve the constraint `field("foo", result) = 3` with `undef` as the previous partial value for result. This gives the value:

`("foo" 3, "bar" undef)`

Note that knowing the type of the tuple involved makes inverting the `field` function possible. Finally, this value is put back into the original partial value for `result`, yielding:

`<("foo" 3, "bar" undef), ("foo" 7, "bar" 6.3)>undef`

Table 5.1: Some set operations over partial values. Here, $*$ is either undef or a non-negative integer, n is a non-negative integer, and x is not undef.

Operation	Argument	Argument	Result
size	S_{undef}		undef
size	S_n		n
member	x	S_*	$\begin{cases} \texttt{true} & \text{if } x \in S \\ \texttt{undef} & \text{otherwise} \end{cases}$
subset	S1	$S2_*$	$\begin{cases} \texttt{true} & \text{if } S1 \subseteq S2 \\ \texttt{undef} & \text{otherwise} \end{cases}$
subset	$S1_*$	$S2_*$	undef
subset	$S1_*$	S2	$\begin{cases} \texttt{false} & \text{if } \neg(S1 \subseteq S2) \\ \texttt{undef} & \text{otherwise} \end{cases}$
union	$S1_*$	$S2_*$	$(S1 \cup S2)_{undef}$
union	S1	$S2_*$	$(S1 \cup S2)_{undef}$
union	$S1_*$	S2	$(S1 \cup S2)_{undef}$
intersection	$S1_*$	$S2_*$	$(S1 \cap S2)_{undef}$
intersection	S1	$S2_*$	$(S1 \cap S2)_{undef}$
intersection	$S1_*$	S2	$(S1 \cap S2)_{undef}$
- (set difference)	S1	$S2_*$	undef
- (set difference)	$S1_*$	$S2_*$	undef
- (set difference)	$S1_*$	S2	$(S1 - S2)_{undef}$

Table 5.2: Some sequence operations over partial values. Here, $*$ is either undef or a non-negative integer, and n, m, and i are non-negative integers. The notation $\texttt{undef}^l$ means a comma separated list of undef's of length l.

Operation	Argument	Argument	Result
length	$\texttt{S1}_{\texttt{undef}}$		undef
length	$\texttt{S1}_n$		n
index	$\texttt{S}_{\texttt{undef}}$	i	$\begin{cases} \texttt{undef} & \text{if } \texttt{i} > \textit{length}(\texttt{S}) \\ \textit{index}(\texttt{S}, \texttt{i}) & \text{otherwise} \end{cases}$
index	$\texttt{S}_n$	i	$\begin{cases} \texttt{error} & \text{if } \texttt{i} > n \\ \texttt{undef} & \text{if } \textit{length}(\texttt{S}) < \texttt{i} \leq n \\ \textit{index}(\texttt{S}, \texttt{i}) & \text{otherwise} \end{cases}$
header	$\texttt{S}_{\texttt{undef}}$		$(\textit{header}(\texttt{S}))_{\texttt{undef}}$
header	$\texttt{S}_n$		$\begin{cases} (\textit{header}(\texttt{S}))_{n-1} & \text{if } \textit{length}(\texttt{S}) = n \\ \texttt{S}_{n-1} & \text{otherwise} \end{cases}$
first	$\texttt{S}_{\texttt{undef}}$		$\begin{cases} \texttt{undef} & \text{if } \textit{length}(\texttt{S}) = 0 \\ \textit{first}(\texttt{S}) & \text{otherwise} \end{cases}$
first	$\texttt{S}_n$		$\begin{cases} \texttt{error} & \text{if } n = 0 \\ \texttt{undef} & \text{if } \textit{length}(\texttt{S}) = 0 \wedge n \neq 0 \\ \textit{first}(\texttt{S}) & \text{otherwise} \end{cases}$
trailer	$\texttt{S}_{\texttt{undef}}$		$\begin{cases} \texttt{S}_{\texttt{undef}} & \text{if } \textit{length}(\texttt{S}) = 0 \\ (\textit{trailer}(\texttt{S}))_{\texttt{undef}} & \text{otherwise} \end{cases}$
trailer	$\texttt{S}_n$		$\begin{cases} \texttt{error} & \text{if } n = 0 \\ \texttt{S}_{n-1} & \text{if } \textit{length}(\texttt{S}) = 0 \wedge n \neq 0 \\ (\textit{trailer}(\texttt{S}))_{n-1} & \text{otherwise} \end{cases}$
last	$\texttt{S}_{\texttt{undef}}$		undef
last	$\texttt{S}_n$		$\begin{cases} \texttt{error} & \text{if } n = 0 \\ \texttt{last(S)} & \text{if } \textit{length}(\texttt{S}) = n \wedge n \neq 0 \\ \texttt{undef} & \text{otherwise} \end{cases}$
||	$\texttt{S1}_{\texttt{undef}}$	S2	undef
||	$\texttt{S1}_{\texttt{undef}}$	$\texttt{S2}_*$	undef
||	$\texttt{S1}_n$	$\texttt{S2}_{\texttt{undef}}$	$\textit{app}(\texttt{S1}, \textit{app}(\texttt{<undef}^{n-\textit{length}(\texttt{S1})}\texttt{>}, \texttt{S2}))_{\texttt{undef}}$
||	$\texttt{S1}_n$	$\texttt{S2}_m$	$\textit{app}(\texttt{S1}, \textit{app}(\texttt{<undef}^{n-\textit{length}(\texttt{S1})}\texttt{>}, \texttt{S2}))_{n+m}$
||	$\texttt{S1}_n$	S2	$\textit{app}(\texttt{S1}, \textit{app}(\texttt{<undef}^{n-\textit{length}(\texttt{S1})}\texttt{>}, \texttt{S2}))_{n+\textit{length}(\texttt{S2})}$
||	S1	$\texttt{S2}_{\texttt{undef}}$	$\textit{app}(\texttt{S1}, \texttt{S2})_{\texttt{undef}}$
||	S1	$\texttt{S2}_m$	$\textit{app}(\texttt{S1}, \texttt{S2})_{\texttt{length(S1)}+m}$

So, the process of constraint solving can be summarized as follows: starting from the innermost observer in the *lhs*, apply the observers to the previous partial value (and then to the result of the previous application) until some application yields `undef`, a partial set value, or in some cases a partial sequence value. Then, use the inverses of rest of the observers to further define the partial value found in the previous step. Finally, replace the partial value from the first step with this more defined (but still possibly partial) value in the original partial value. The keys to the correctness of this kind of constraint solving are that the (partial) value produced must satisfy the constraint, and that the value produced must also satisfy all previous constraints used to define that value. In order to maintain this second property, the constraint solver can never remove an element from the defined subset of a partial set value, and can never change any defined parts of partial sequence, tuple, and object values. Of course, backtracking can result in this property being violated. However, the property must hold for any successful execution path.

As the inverses of the observers are not functions, applying these inverses often gives a partial value that represents a (possibly infinite) set of total values. For example, consider the constraint `header(result) = <3>` with `undef` as the previous partial value for result. Applying the inverse of `header` yields $\texttt{<3>}_2$ as the value of `result`, which represents the infinite set of sequences of integers of length 2 whose first element is 3. The inverse of `trailer` is analogous. And, of course, the inverse of `index` depends on its integer argument. For example, solving the constraint `index(result, 3) = 4` with `undef` as the previous partial value for `result` gives $\texttt{<undef, undef, 4>}_{\texttt{undef}}$. Figure 5.7 demonstrates the use of the inverses of all three of these observers.

While the *lhs* is used in the same way for any kind of constraint in constraint solving, the actual (partial) value produced depends on whether the constraint is a `length`, `size`, `member`, `subset`, or `=` constraint. (See Section 5.4.1 for a discussion of the kinds of constraints.) The `length` and `size` constraints on sequences and sets, respectively, cause the appropriate partial set or sequence value to become total when it grows to the size given in the constraint. Note that sequences can still contain partial values, even when the total size of the sequence is known. Knowing the length of partial sequence values is significant because constraints whose *lhs* uses

`last` can not be applied until the length of the constrainted sequence is known. For example, a constraint like `field("ord", last(seqOrderedPair'))` = 3 is delayed until the length of `seqOrderedPair'` is known. If this length never becomes known, the execution procedure assumes that the current length of the sequence is its total length (unless it has length zero) and attempts to apply any `last` constraints. If this fails (because the last element of the defined prefix is inconsistent with the constraint), then the execution procedure backtracks and assumes that the total length of the sequence is one greater than its current length. This guarantees that the constraint can be solved (if it has the correct type), because no previous constraints have constrained anything outside the defined prefix of the partial sequence value. Knowing whether a set value is partial or total is important in determining how to solve **member** and **subset** constraints on that value, as described next.

The solving of the **member** and **subset** constraints depends on whether they constrain total or partial set values. In the first case, the complete set value is already known, and so the execution technique need only check that the constraint is consistent with that value in the obvious way. Otherwise, the partial set value is further defined by unioning it with the singleton set constructed from the value in the constraint for **member** constraints, or by unioning it with the set value contained in the constraint for **subset** constraints. This latter case can cause the partial set value to become larger than the size specified by a previous `size` constraint, and so backtracking can occur. Since solving either **member** or **subset** constraints can cause the defined subset of a partial set value to grow, either can cause this partial set value to become total if its size is known.

As = constraints directly define (part of a) value, all that need be done is to check that any partial value constructed so far is consistent with the value given by the constraint. For example, if the current (partial) value for `S'` is `<1, undef, 3>`$_{\texttt{undef}}$ and the constraint being applied is `S' = <1, 5, 3, 7>`, the partial value and the constraint are consistent, and the value of `S'` becomes completely known. If this consistency check fails, backtracking occurs.

In the presence of aliasing, two constraints with completely different *lhs*'s might constrain the same value. To account for this, values contained in objects must be manipulated indirectly, so that any change is visible everywhere that the object

occurs. For example, suppose that `x` and `y` denote the same object. Then the post-condition `x' = 3 /\ y' = 4` is unsatisfiable, and the execution technique should report an error. In executing this post-condition, the constraints `x' = 3` and `y' = 4` are generated. Whichever is solved first gives a total post-state value for the object, and so attempting to solve the second constraint causes an error, as the post-state value proposed for the object is inconsistent with the value it already has.

If a type error occurs while solving constraints, the execution procedure backtracks on the chance that the use of a union type has created a backtrack point earlier in the execution. For example, given the partial value

$$\texttt{<("fst" 2, "snd" undef), undef>}_5$$

as the current post-state value for `seqTuple`, trying to apply the constraint

```
field("thrd", index(seqTuple', 4)) = 7
```

causes a type error, and backtracking occurs. This backtracking has a chance of succeeding, for example, if `seqTuple'` is of a union type where one type in the union is a tuple type with `"fst"` and `"snd"` fields, and another type in the union is a tuple type with `"fst"`, `"snd"`, and `"thrd"` fields. In this case, the first appearance of a constraint on a value of the union type would create a backtrack point that would allow all the types in the union to be tried if necessary.

Finally, if some parts of post-state object, sequence, or tuple values remain undefined after all constraints have been generated and applied, then the frame axioms of SPECS-C++ imply that these parts should be whatever they were in the pre-state. Roughly speaking, frame axioms are used to ensure that no part of the state changes unless it must change to satisfy the post-condition. Thus, when a pre-state value exists and the corresponding post-state value is partial after all constraints have been applied, then the missing parts of the post-state value are "filled in" with the corresponding parts of the pre-state value. For sequences, this means that the length of the sequence and the value contained at any index position does not change from the pre- to the post-state unless the post-condition forces change. For tuples, each field does not change from the pre- to the post-state unless forced to. Similarly, the value contained in an object does not change unless mandated by the post-condition.

```
int operator++(int& x);
  modifies x
  post: x' = x^ + 1 /\ result = x'
```

Figure 5.4: The specification of a pre-increment function. The SPECS-C++ keyword `result` is used to refer to the result of the function.

After applying the frame axioms, set and sequence values that are still partial are made total, except that sequences are still allowed to contain partial values. At this point, it may be possible to generate further constraints from the delayed portions of the post-condition, so the execution procedure iterates. The execution algorithm may return partial tuple, sequence, and object values if no corresponding pre-state value exists. As most model-based specification languages do not have implicit frame axioms, this step would be omitted in executing such languages.

5.4.3 Examples

Figure 5.4 shows a specification of the built-in C and C++ prefix increment function. The first conjunct of the post-condition is immediately evaluated into a constraint. Both sides of the = in the second conjunct are unknown, so it is delayed. After the constraint derived from the first conjunct has been solved, the value of `x'` is known, and so the second conjunct can be evaluated into a constraint, which is then solved to give a value to `result`.

Consider the specification of a sorting procedure of Figure 5.5. This example is similar to that presented in Figure 5.3, except that this sorting procedure allows duplicates in the sequence to be sorted, and also uses the abstract functions `sorted`, `count`, and `sameCountAs` to modularize and to improve the readability of the specification. In the execution of the post-condition, the first and second conjuncts produce constraints that are solved into a sequence that contains the same elements as the pre-state value of `si`. The third and fourth conjuncts are then evaluated over that

```
define sorted(seqint s) as bool s.t.
  sorted(s) = \forall int i [1 <= i <= length(s) =>
                              index(s, i - 1) <= index(s, i)]

define count(int i, seqint s) as int s.t.
     (s = <> => count(i, s) = 0)
  /\ (s != <> =>
        ((first(s) = i => count(i, s) = 1 + count(i, trailer(s)))
      /\ (first(s) != i => count(i, s) = count(i, trailer(s)))))

define sameCountAs(seqint s1, s2) as bool s.t.
  \forall int x [x member s2 => count(x, s2) = count(x, s1)]

void sort(seqint& si);
  modifies: si
  post: \forall int x [ x member si^ =>
           \exists int i [1 <= i <= length(si^) /\
                          index(si', i) = x]]
     /\ length(si') = length(si^)
     /\ sorted(s')
     /\ sameCountAs(s^, s')
```

Figure 5.5: The specification of a sort procedure that can handle duplicates. Here, seqint is again the type of sequences of integers.

```
Int_Set RelTo(Relation theRel, int key);
 pre: \exists (OrderedPair p) [
        (p \in theRel) /\ field("first", p) = key]
 post: \forall (OrderedPair p) [
        (p \in theRel) /\ field("first", p) = key =>
          field("second", p) \in result
      ] /\
      \forall (int i) [
        (i \in result) =>
          \exists (OrderedPair p) [
            (p \in theRel) /\ i = field("second", p)
            /\ key = field("first", p)]]
```

Figure 5.6: The specification of a "related to" function for relations. `Relations` are modeled as sets of `OrderedPairs`, and `OrderedPairs` are tuples with `"first"` and `"second"` fields. `Int_Set` is the type of sets of integers.

sequence to check that it is a permutation of the pre-state value of `si` and that it it sorted. If so, then the execution is complete. Otherwise, the execution procedure backtracks and a different sequence is tried. Note that the order of the conjuncts does not matter to the execution procedure, as the calls of `sorted` and `sameCountAs` are delayed until a total value for `si'` has been found.

Figure 5.6 is the specification of a function that returns everything in an integer relation that is related to the integer argument `key`. The pre-condition specifies that something must be related to `key`. The first conjunct of the post-condition is immediately evaluated into constraints, and these constraints are solved into a partial value for `result`. As this value is partial, the second conjunct can not be evaluated, and so no more constraints can be found. Next, the SPECS-C++ frame axioms are applied to the partial value for `result`. For sets, this just results in the partial value being made total. This allows the second conjunct of the post-condition to be

```
seqint constrainseq();
 post: index(header(header(trailer(trailer(result)))), 2) = 3
```

Figure 5.7: A specification that produces only one constraint on `result`. Type `seqint` is the type of sequences of integers.

evaluated, and it is satisfied as the execution procedure does not put any "extra" elements into the value of `result`.

Finally, the example of Figure 5.7 demonstrates the potential complexity of constraints and the amount of information that can be derived from such constraints. The post-condition is in the form of a constraint, and so is immediately evaluated into one. This constraint is solved into the following partial sequence value.

$$\texttt{<undef, undef, undef, 3, undef, undef>}_{\texttt{undef}}$$

Note that the partial sequence value has two trailing `undefs`. This represents the knowledge (derived from the constraint) that the total sequence must have a length of at least six, as the *lhs* could not be applied to any shorter sequence. However, no unnecessary `undefs` are added. Whenever the execution technique is able to find a post-state set or sequence value, it always produces the smallest sequence or set value that satisfies both the post-condition and the frame axioms.

5.4.4 Comments on the Implementation

The execution procedure is currently implemented as an interpreter for SPECS-C++ specifications. The interpreter uses an abstract syntax for SPECS-C++ that is basically a parse tree representation of the concrete syntax presented in the examples. The interpreter consists of 2800 lines of Standard ML [Pau91] code, and was written with the aid of the literate programming tool Noweb [Ram91]. Backtracking is implemented via continuations. As in denotational semantics [Sch86], state is im-

plemented via a store that is passed to and returned from all functions that modify the state.

5.4.5 Performance

The performance of the execution algorithm depends on the structure of the pre- and post-conditions of the operation being executed and on the pre-state values involved. As pre-conditions are just evaluated for their boolean value, all that is involved is evaluating assertions. In executing post-conditions, some parts of the post-condition are just evaluated, while other parts are used in generating constraints. Hence, the time needed for generating and solving constraints must also be considered.

The time used for directly evaluating assertions is polynomial in the size of the values involved in the assertion, as the algorithms implementing the most expensive of the built-in SPECS-C++ operations (union, intersection, difference, subset) are $O(n^2)$ for sets of size n. The time for executing abstract and member function calls in assertions is just the time for evaluating the arguments and the time for evaluating the function body. As the body of a quantified assertion is potentially executed once for each element of the domain quantified over, the execution time is linear in the size of that domain, and polynomial in the number of nested quantifiers. Otherwise, the evaluation is linear in the number of other boolean connectives.

The time needed generating constraints is largely determined by how much backtracking occurs, and so is exponential in the worst case. (Actually, backtracking can occur both in generating and in solving constraints. The time used in backtracking is included in the time for generating constraints because almost all of the backtrack points, i.e. the points in the execution that can be "backtracked to", are created in constraint generation. The only other backtrack points are created in constraint solving for solving constraints over union types and a few forms of constraints on sequences.) Because of backtracking, the time needed for generating constraints is exponential in the number of disjunctions and implications in the post-condition, as well as in the size of the domain of each existentially quantified assertion. As no backtrack points are created in generating constraints from universally quantified assertions, the time needed is just linear in the size of the domain quantified over.

As in evaluating assertions, the time for of generating constraints is polynomial in the depth of nested quantifiers. Finally, the execution time is linear in the number of other boolean connectives. If, as often happens in executing specifications, no backtracking occurs in generating constraints, then the execution time is just linear in the number of disjuncts and implications, and linear in the size of the domain of existentially quantified assertions.

The time for solving constraints depends on the number of constraints, the kind of constraint (=, **member**, **subset**, ...), and the structure of the *lhs* of the constraint, but is polynomial in the number of constraints in the worst case. Redundant constraints impose the overhead of consistency checking with the partial value so far constructed. As the time used for consistency checking is generally less than that for constraint solving, we concentrate on the time for solving constraints. Ignoring for the moment the time cost imposed by the structure of the *lhs*, a = constraint can be solved in constant time, as the value needed is contained directly in the constraint. Assuming that the size of the *val* part of all **subset** constraints is bounded by some small constant, the time for solving **member** and **subset** constraints is linear in the number of **member** and **subset** constraints that constrain the same set value, as the number of these constraints bounds the size of the set produced, and the constraint solver checks for duplicate elements in producing set values. Thus, the total time for solving the n **member** and **subset** constraints that constrain a single set value is $O(n^2)$.

The time cost imposed by the structure of the *lhs* depends on the number and kind of observers it contains. The most expensive observer is **index**, as it requires either navigating within the partially defined sequence value, or allocating new index positions for that value. Either operation is directly porportional to the index of the position constrained. If a sequence value is completely constrained by n constraints with **index** in the *lhs*, then the maximum length of the sequence is n. Hence, for a constraint with m observers, the maximum time is $O(mn)$. As there are n constraints, the total maximum time for solving these constraints is $O(mn^2)$. As the number of constraints is typically much greater than the number of observers appearing in any *lhs*, $O(n^2)$ is a fair upper bound. Combining this with the time needed to solve a constraint without considering the structure of the *lhs*, the maximum time for solving

a set of n constraints is $O(n^3)$. Note, however, that this worst case only applies to solving constraints on values that are sequences of sets, as only such values can be specified by `member` constraints with `index` appearing in the *lhs*. For values of other types, the worst case time for constraint solving is $O(n^2)$.

In general then, for post-conditions that force a great deal of backtracking, this backtracking dominates the total execution time for the algorithm, and the execution time can be as bad as exponential in the larger of either the maximum size of a domain of an existentially quantified assertion or the total number of disjuncts and implications. More typically, the execution of the post-condition will not require significant backtracking, and in that case the execution time is polynomial. The number of constraints generated from a post-condition is (at most) linear in the number of conjuncts, linear in the size of the domains of any universally quantified assertions, and polynomial in the depth of nested universal quantifiers. Thus, for a post-condition with m conjuncts in which the largest domain of a universally quantified assertion is of size n and the maximum depth of universal quantifier nesting is l, the number of constraints generated is $O(m+n^l)$. Combining this with the time needed for solving constraints, the maximum time needed for executing a post-condition without backtracking is $O((m+n^l)^3)$. As the time used in generating constraints is proportional to the maximum number of constraints generated, the worst case time for solving constraints dominates the worst case time for generating them, and so bounds the execution time. In practice, the time needed to execute a specification (again, in the case where significant backtracking is not required) rarely approaches this upper bound.

5.4.6 Limitations

The most significant limitation is the non-executability of quantified assertions in which the bound variable ranges over an unspecified or infinite domain. As evidenced by **fase3** [KK93, Kra88], some progress on executing such assertions is possible, but even in **fase3** these assertions can only be executed if the bound variable implicitly ranges over a finite domain. Similarly, there are no plans to make specifications involving infinite sets and sequences executable. As constraint solving over real valued variables in the sense of CLP($\mathcal{R}$) [JMSY92] and Prolog III [Col90] does not seem to

be very helpful in executing typical model-based specifications, there are no plans to extend the execution technique with more general arithmetic constraint solving. However, further research is planned on removing some of the current limitations of the execution technique. This research is outlined in the next section.

5.4.7 Future Work

Several relatively simple extensions of the execution technique would enhance its usefulness. As backtracking is already built into the interpreter, it would not be difficult to return multiple post-state values for the same object (or multiple function results) that satisfy the post-condition. This would be useful for underdetermined or nondeterministic specifications. For example, given the postcondition:

```
x' = 3 \/ x' = 4
```

both 3 and 4 are possible post-state values for x, and returning both would be desirable. Another simple extension is the execution of instantiations of generic specifications. Generic specifications are specifications parameterized by types, and instantiation of a generic specification occurs when an actual type is substituted for each parameter type. As the actual parameter type is known at execution time, the execution technique would need only minor changes to execute such specifications.

More interesting future work centers on the use of partial values in executing specifications. Currently, quantified assertions are only evaluated or used to generate constraints if the domain that the bound variable ranges over completely known – that is, if the domain is not a partial value. However, evaluating a quantified assertion over a partial set or sequence value (i.e. the bound variable is restricted to range over the partial set or sequence) could detect universally quantified assertions that are false, and existentially quantified assertions that are true. Any constraints generated from such an assertion would be valid. However, the assertion would have to be reevaluated for constraints whenever more elements of the domain become known, and so many redundant constraints would be generated. Hence, more research is needed before such an extension to the execution technique is implemented.

Another interesting problem is the use of partial values as arguments in abstract or member function calls. One aspect of this problem is the execution of calls of recursive abstract functions where the recursion depends on an actual argument that is a partial value. For example, consider executing the abstract function count from (Figure 5.5) when the length of the sequence argument is not known. The problem is to find a reasonable way to ensure that such a call terminates. On the other hand, abstract function calls with partial valued arguments can also be viewed as constraints on those arguments. For example, a natural post-condition for a sort procedure, where S is an object containing a sequence value, is:

```
permutation(S^, S') /\ sorted(S')
```

Executing this post-condition would require that the call to **permutation** be used to generate candidate values for S', which does not occur with the current execution technique. The major difficulty is to determine whether an abstract function call with partial-valued arguments should be used for generating constraints (as in the example call to **permutation**, or whether it should be delayed until the partial valued arguments become total (as in the call to **count**). In general, this problem is probably undecidable (as it seems to rely on determining whether a function terminates when called with the given arguments), but better use of these calls than that made by the current execution technique should be possible.

5.5 Comparison of the Execution Technique with CLP

The most obvious difference between our execution technique and CLP languages is the lack of constraint solving on the real numbers. While such constraint solving could be added to the interpreter in a fairly orthogonal manner, the utility of doing so is questionable as specifications written in model-based specification languages rarely produce such constraints. On the other hand, our execution technique deals with a richer set of non-arithmetic domains than most CLP languages and provides different constraints that are more suited to specifications over such domains. In particular, **member** and **subset** constraints over sets provide a level of expressiveness not found in most CLP languages.

Our execution technique handles backtracking differently than logic and CLP languages. In such languages, backtracking can occur whenever multiple rules could be applied at some point in program execution, and, given repeated failure, the execution will eventually backtrack to all such rules whether they have a chance of leading to success or not. Our technique explicitly checks that an alternative has a chance of succeeding before creating a "backtrack point" that allows backtracking to that alternative. As previously mentioned, this checking is done by evaluating alternatives in a non-strict manner to see if any must be false, just from the values that are already known. We have found that for a large majority of specifications (even specifications written without executability in mind) this approach eliminates any possibilities for backtracking. The specification of an absolute value procedure of Figure 5.2 is typical. Clearly, this leads to more efficient execution.

Our technique shares the concept of delaying constraints with Prolog III and CLP($\mathcal{R}$), but differs in how constraints are delayed. In Prolog III, constraints are only delayed explicitly (by the programmer), while in our technique, the delaying of constraints is always transparent to the user. CLP($\mathcal{R}$) also uses implicitly delayed constraints, but the only delayed constraints are nonlinear equations and inequalities. In our technique, delayed constraints consist both of constraints involving `last` applied to partial sequences of unknown length and of any assertion that can't be evaluated or converted to constraints until more post-state values are known.

This is related to the use of logical variables, as it is our avoidance of logical variables that causes some forms of this second kind of delayed constraint. Consider the example of Figure 5.4, and recall that the second conjunct of the post-condition was delayed until the value of `x'` was known. Given the obvious CLP program analogous to this postcondition with Xpre, Xpost, and Result as logical variables representing the pre- and post-state values of `x` and the post-state value of `result`, respectively, a CLP interpreter would produce the following constraint set.

$$\{\mathsf{Xpost} = \mathsf{Xpre} + 1,\ \mathsf{Result} = \mathsf{Xpost}\}$$

For any particular value for Xpre, say 3, this would be (iteratively) simplified into the following answer constraint set.

$$\{\mathsf{Xpost} = 4,\ \mathsf{Result} = 4\}$$

In fact, whenever logical variables can eventually be bound to actual values, our execution technique produces the effect of logical variables, and the main difference from an execution standpoint is whether iteration occurs while finding constraints or while simplifying them.

Given this, it is reasonable to ask why we did not use logical variables to represent post-state values in our execution technique. Our avoidance of logical variables is based on the kind of answer that is usually wanted when a specification is executed. For example, one use of an executable specification is to "fill in" for the corresponding implementation in the use or testing of code that uses that implementation. This dependent code can not deal with answer constraints or with logical variables contained in the values it receives from the executable specification. However, it can deal with parts of these values being undefined. In C++, for example, this is equivalent to an array with uninitialized index positions, or to a struct with uninitialized fields.

Another possibility would be to use logical variables inside the execution procedure, and then to replace any logical variables that cannot be substituted out with `undef` before any values produced are made available to outside code. As previously noted, this substitution process requires the same kinds of iteration that occur in the current execution procedure. Additionally, any implicit constraints found through the use of logical variables – for example, that the first and second index positions of a sequence contain the same logical variable `X` and so the same value, where no binding for `X` exists in the associated substitution – would be lost in the replacement of logical variables with `undef`.

Unlike CLP($\mathcal{R}$), our execution technique does not use the standard unification algorithm. This is again related to our not using logical variables, as unification is normally used to produce bindings for logical variables. However, our techniques for applying constraints to partial values and for applying the SPECS-C++ frame axioms can be thought of as variants of unification that produce values instead of substitutions.

While Prolog and most CLP languages are untyped, SPECS-C++ specifications have explicit type information, and this information can be exploited in constraint solving. In particular, type information is useful in constructing tuples (as the type of the tuple gives the number and type of its fields) and in handling union types

sensibly. The existence of union types justifies backtracking on type errors, as the problem may have been caused by using the wrong type from the union. Having the explicit union type limits the amount of backtracking that can occur, as most unions contain only a small number of types. In Prolog III, backtracking occurs when no more rules can be applied, which could be caused by the equivalent of a type error, or simply by unsatisfiability. This type of backtracking has no explicit bound, and so is a potential source of inefficiency.

Another difference is language syntax. While logic and CLP languages use a rule format that facilitates execution, an execution technique for model-based specification languages must start with the syntax of such languages. In the literature on the execution of model-based specifications, three distinct approaches to dealing with specification language syntax dominate:

1. Restrict the executable subset of the language so that specifications can be directly executed. In particular, this usually means that references to post-state values can not appear in the bodies of quantified assertions, and that only assertions of the form `x' = val` can be used to provide post-state values, where `val` is strictly an expression over pre-state values. This is the most prevalent approach, and examples include EPROL [HI88, HI86], *me too* [Hen86], *SMLVIEW* [O'N92a, O'N92b], and the technique used for executing IPTES mini-specifications [RE93, LL91, AELL92].

2. Use Prolog syntax or code in the specification language, and execute specifications via Prolog. Examples include PLEASE [TC89] and OBSERV [TY92].

3. Explicitly translate specifications to Prolog programs. A number of researchers have experimented with this approach [DKC90, WE92], but to the authors' knowledge, no completely automated translator exists.

Thus, our approach of deriving constraints from specifications and then applying CLP-like techniques seems to be unique, and to supply a unique combination of syntax that is natural for specifiers and a very high level of executability.

In conclusion, we have described an execution technique for model-based specification languages with a number of unique and useful features. On the executable specification side, this technique provides:

- Handling of mutation and aliasing at the (executable) specification level. This permits the writing of practical, "industrial strength" specifications of implementations that rely on these features.

- Execution of high level specifications. Such highly abstract specifications would occur early in the software development process, and so the execution of these specifications brings validation activities to the earliest stages of development.

In the application of CLP techniques, our work demonstrates:

- Checking that backtracking to an alternative would lead to a possiblility of success before creating the backtrack point.

- Constraint solving in presence of type information. In particular, we have shown how to use backtracking to deal with union types sensibly.

- Application of CLP techniques to new domains, such as objects, sets, sequences, and tuples.

- Automatic generation of constraints. This is necessary for the practical application of CLP techniques to executing specifications, and may provide a basis for CLP languages with more human-oriented syntax.

Thus, our work has practical implications for software development and opens the gateway into a new application arena for constraint logic programming.

6. GENERAL SUMMARY

6.1 Discussion of Results

As described in the introduction of this dissertation, the focus of the research described here has been to develop an execution technique capable of executing specifications written at the level of abstraction that typifies non-executable, model-based specifications. The execution technique for model-based specifications largely achieves this goal. The first efforts in developing the execution technique led to a prototype interpreter for DFD-SPECS specifications. The main contributions of this interpreter are a practical operational semantics for formalized data flow diagrams and the first work on executing assertions over model types.

After the realizations that a better technique for executing assertions was needed, and that this technique would be directly applicable in executing other model-based specification languages, work began on an interpreter for SPECS-C++. Changing the executed specification language was important in isolating parts of the execution technique that are common to model-based specification languages and parts that are specific to a particular language. Research into executing assertions led to the ability to directly execute quantified assertions, and to the use of quantified for generating constraints on post-state values. Constraint solving was also introduced into the interpreter, but the relationship with modern constraint logic programming techniques was never considered, and so the constraint solving was correspondingly crude. As SPECS-C++ differs from DFD-SPECS in allowing the specification of side effects, the interpreter was extended to execute such specifications. As variables could now change value from the pre-state to the post-state, frame axioms became important. This led to the formalization of the operational semantics of SPECS-C++ frame axioms, and incorporation of this semantics into the interpreter. Although this

interpreter can not execute many of the specifications that are executable by the current incarnation of the execution technique, it does have one major advantage over more recent developments: the time required for executing a particular specification is predictable, and can easily be estimated by inspecting the specification.

While the interpreter could now execute a large and useful subset of SPECS-C++, it still had a number of difficiencies. Research on the design of SPECS-C++ itself led to the incorporation of an elegant and uniform theory of objects into the language. Because of aliasing among objects, and because objects can contain other objects, the interpreter was not sufficient for executing SPECS-C++ with objects. Additionally, the interpreter dealt poorly with underdetermined (nondeterministic) specifications, and could not execute incomplete specifications. Research into constraint logic programming techniques led to the use of backtracking for dealing with underdetermined specifications, and to the development of a theory of partial values so that incomplete specifications could be executed. Partial values also allowed much more sophisticated constraint solving over SPECS-C++ domains, as constraints could be applied individually and in any order to produce partial values. This refined constraint solving proved to be a natural vehicle for executing specifications involving objects, as a partial value could be contained in an object, and later constraints on objects aliased with that object could easily be used to further define that partial value and could be checked for consistency with the partial value. Thus, the use of techniques inspired by constraint logic programming has greatly enhanced the expressiveness of executable SPECS-C++ specifications. Unfortunately, this additional expressiveness was not achieved without cost, as the use of backtracking has made execution times for specifications harder to predict. On the other hand, the current execution technique does not backtrack when executing any specification that could be executed by the previous interpreter. Thus, the gain in expressiveness has not resulted in a loss of efficiency.

Applying the execution technique in executing DFD-SPECS and SPECS-C++ has allowed the investigation of the executability of existing specifications written in these languages, which is valuable because these specifications were written without executability in mind. The body of specifications in question were developed as examples for the undergraduate and graduate software engineering courses at Iowa State

University. Typical examples of these specifications include the DFD-SPECS spelling checker specification (Figure 3.1) and the SPECS-C++ specifications of ADTs Table (Section 2.2.3) and Bounded List (Section 2.3.3). Of these, the majority (approximately 70 percent) are executable as written. Almost all of the remaining examples are executable with very minor modifications that do not greatly reduce the level of abstraction present in the specification. The refinement of the execution technique with techniques from constraint logic programming makes even more specifications executable, especially underdetermined and nondeterministic specifications. As the specifications used as examples for students tend to be complete and deterministic, they often do not require the full power of the refined execution technique. However, comparison with specification examples from the literature on specification languages that use logic programming techniques for executability revealed the power and utility of the refined execution technique. Typical examples of these specifications include the sorting examples of Chapter 5 (Figures 5.3 and 5.5).

Thus, the current execution technique can execute a large and useful subset of formal, model-based specification languages such as DFD-SPECS and SPECS-C++. As specifications written at a high level of abstraction can be executed, the execution technique provides a way to gain the benefits of executable specifications without losing the benefits of traditional non-executable specifications. Thus, the execution technique should greatly enhance the usefulness of formal specifications in industrial software development settings. The execution technique also introduces constraint solving in new domains, and the use of constraint logic programming techniques in a new application area. Hence, the execution technique has important implications both in practical software development and in applying constraint logic programming to real world problems.

BIBLIOGRAPHY

[AELL92] Michael Andersen, René Elmstrøm, Poul Bøgh Lassen, and Peter Gorm Larsen. Making Specifications Executable – Using IPTES Meta-IV. In *Euromicro '92*, September 1992.

[AJ88] L. Augustsson and T. Johnsson. *Lazy ML User's Manual.* Department Of Computer Science, Chalmers University of Technology, Gőteborg, January 1988.

[Bak93] Albert L. Baker. Computer Science 411 Class Notes. SPECS-C++ as taught in an undergraduate software engineering course, Iowa State University, Ames IA, 1993.

[BL90] V. Berzins and Luqi. Languages for Specification, Design, and Prototyping. In P. A. Ng and R. T. Yeh, editors, *Modern Software Engineering: Foundations and Current Perspective*, chapter 4, pages 83 – 118. Van Nostrand Reinhold, New York, 1990.

[BM93] Paulo Borba and Silvio Meira. From VDM Specifications to Functional Prototypes. *The Journal of Systems and Software*, 21(3):267 – 278, June 1993.

[BMR93] Alex Borgida, John Mylopoulos, and Raymond Reiter. ... And Nothing Else Changes: The Frame Problem in Procedure Specifications. In *Proceedings Fifteenth International Conference on Software Engineering*, Baltimore, May 1993.

[Boe81] Barry W. Boehm. *Software Engineering Economics.* Advances in Computing Science and Technology. Prentice Hall, Englewood Cliffs, N.J., 1981.

[BY91] J.M. Bieman and H. Yin. Monitoring the Correctness of Software. In *Proc. ISMM Int. Symp. on Engineering and Industrial Applications*, pages 79–82, December 1991.

[BY92] J.M. Bieman and H. Yin. Designing for Software Testability Using Automated Oracles. In *Proc. International Test Conference*, pages 900–907, September 1992.

[CDD+90] D. Carrington, D. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An Object-Oriented Extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296, Amsterdam, 1990. Elsevier Science Publishers (North-Holland).

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, second edition, 1984.

[Coh85] Jacques Cohen. Describing Prolog by Its Interpretation and Compilation. *Communications of the ACM*, 28(12):1311–1324, December 1985.

[Coh90] Jacques Cohen. Constraint Logic Programming Languages. *Communications of the ACM*, 33(7):52 - 68, July 1990.

[Col85] Alain Colmerauer. Prolog in 10 Figures. *Communications of the ACM*, 28(12):1296 - 1310, December 1985.

[Col90] Alain Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69 - 90, July 1990.

[Col91] David L Coleman. *Formalized Structured Analysis specifications.* PhD thesis, Iowa State University, Ames, Iowa, 50011, 1991.

[DFPT90] M. Degl'Innocenti, G. L. Ferrari, G. Pacini, and F. Turini. RSF: a Formalism for Executable Requirements Specifications. *IEEE Transactions on Software Engineering*, 16(11):1235 - 1246, November 1990.

[DKC90] A.J.J. Dick, P.J. Krause, and J. Cozens. Computer Aided Transformation of Z into Prolog. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1989*, Workshops in Computing, pages 71–85, Berlin, 1990. Springer-Verlag.

[EHMO91] G. W. Ernst, R. J. Hookway, J. A. Menegay, and W. F. Ofgen. Modular Verification of Ada Generics. *Computer Languages*, 16(3/4):259–280, 1991.

[ES90a] Susan Eisenbach and Chris Sadler. Functional Programming on Parallel Architectures. In Darrel Ince and Derek Andrews, editors, *The Software Life Cycle*, pages 107 - 126. Butterworth & Co. Ltd., London, 1990.

[ES90b] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[Fai85] R Fairley. *Software Engineering Concepts.* McGraw-Hill, New York, 1985.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* The MIT Press, Cambridge, Massachusetts, 1992.

[GB94] M. Gurski and A. L. Baker. Testing SPECS-C++: A First Step in Validating Distributed Systems Specifications. In *Proceedings of the ISMM International Conference on Intelligent Information Management Systems*, pages 105 - 108, Washington, D.C., June 1994.

[GHG+93] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, 1993.

[GHT84] H. W. Glaser, C. L. Hankin, and D. Till. *Principles of Functional Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1984.

[GHW85] John V. Guttag, James J. Horning, and Jeannette M. Wing. The Larch Family of Specification Languages. *IEEE Software*, 2(4):24 - 36, September 1985.

[GMP92] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal Verification of Ada Programs. In Ursala Martin and Jeanete M. Wing, editors, *First International Workshop on Larch, Dedham 1992*, pages 104–141. Springer-Verlag, 1992.

[GRW91] J. Győrkős, I. Rozman, and T. Welzar. Activation and Validation of the System Specifications. *Microprocessing and Microprogramming*, 31:59 – 64, April 1991.

[Har92a] Andrew Harry. The Application of Programming Languages to the Production of Reference Implementations. Technical Report DITC 209/92, National Physical Laboratory, Teddington, Middlesex TW11 OLW, United Kingdom, October 1992.

[Har92b] Andrew Harry. State of the Art Techniques in Automatic Generation of Reference Implementations and Test Code from Formal Specifications. Technical Report DITC 207/92, National Physical Laboratory, Teddington, Middlesex TW11 OLW, United Kingdom, October 1992.

[Haß87] Manfred Haß. Development and Application of a Meta IV compiler. In D. Bjørner, C. B. Jones, M. Mac an Airchinnigh, and E. J. Neuhold, editors, *VDM '87 - A Formal Method at Work*, pages 118–140. Springer-Verlag, 1987. LNCS 252.

[Hay87] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, Englewood Cliffs, N.J., 1987.

[Hen86] Peter Henderson. Functional Programming, Formal Specification, and Rapid Prototyping. *IEEE Transactions on Software Engineering*, SE-12(2), February 1986.

[HI86] Sharam Hekmatpour and Darrel C. Ince. A Formal Specification-Based Prototyping System. In D. Barnes and P. Brown, editors, *Software Engineering 86*, pages 317 - 335. Peter Peregrinus Ltd., London, UK, 1986.

[HI88] Sharam Hekmatpour and Darrel C. Ince. *Software Prototyping, Formal Methods, and VDM.* Addison-Wesley, Wokingham, England, 1988.

[HJ89] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE, Software Engineering Journal*, 4(6):320–338, November 1989.

[HP87] D. Hatley and E. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, New York, 1987.

[IWH86] Darrel Ince, Mark Woodman, and Sharam Hekmatpour. The Application of Some Artificial Intelligence Tools and Techniques in Software Engineering. In D. Ince, editor, *Software Engineering: The Decade of Change*, pages 81 - 99. Peter Peregrinus Ltd., London, UK, 1986.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339 - 395, July 1992.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[Jon92] K. D. Jones. A Semantics for Larch/Modula-3 Interface Language. In Ursala Martin and Jeanete M. Wing, editors, *First International Workshop on Larch, Dedham 1992*, pages 142–158. Springer-Verlag, New York, 1992.

[JS90] M. Johnson and P. Sanders. From Z Specifications to Functional Implementations. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1989*, Workshops in Computing, pages 86–112, Berlin, 1990. Springer-Verlag.

[KK93] Samuel Kamin and Tim Kraus. Executable Specifications of C++ Classes. submitted for publication, 1993.

[Kle52] S. C. Kleene. *Introduction to Metamathematics.* Van Nostrand, New York, 1952.

[Kra88] Tim Kraus. The FASE3 System for Executable Data Type Specification. Master's thesis, University of Illinois, Urbana, Illinois, 1988. Technical Report 87-1789.

[Kun91] Chenho Kung. Process Interface Modeling and Consistency Checking. *The Journal of Systems and Software*, 15(2):185 - 191, May 1991.

[Lam89] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[Laz89a] Gregory L. Lazarev. Executable Specifications with Prolog. *Dr. Dobb's Journal*, October 1989.

[Laz89b] Gregory L. Lazarev. *Why Prolog?* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[LB88] Jacek Leszczylowski and James M. Bieman. Growing Executable Specifications Using PROSPER. Technical Report TR 88-1, Department of Computer Science, Iowa State University, Ames IA, January 1988.

[LB89] J. Leszczylowski and J.M. Bieman. PROSPER: A Language for Specification by Prototyping. *Computer Languages*, 14(3):165–180, 1989.

[LC93] Gary T. Leavens and Yoonsik Cheon. *Larch/C++ Reference Manual.* Department of Computer Science, Iowa State University, Ames, Iowa 50011, September 1993. available via anonymous ftp from ftp.cs.iastate.edu.

[Lea93] Gary T. Leavens. Inheritance of Interface Specifications (Extended Abstract). Technical Report 93-23, Iowa State University, Department of

Computer Science, September 1993. Appears in the Workshop on Interface Definition Languages, WIDL '94. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[Lel88] Wm Leler. *Constraint Programming Languages.* Addison-Wesley, Reading, Massachusetts, 1988.

[Ler91] Richard Allen Lerner. *Specifying Objects of Concurrent Systems.* PhD thesis, Carnegie Mellon University, Pittsburg, PA, 1991.

[LH92] K. C. Lano and H. P. Haughton. *The Z^{++} Manual.* Lloyd's Register of Shipping, 29 Wellesley Road, Croydon CRO 2AJ, UK, 1992.

[LL91] Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *VDM '91: Formal Software Development Methods*, Berlin, March 1991. VDM Europe, Springer-Verlag.

[LVY88] Luqi, V. Verzins, and R Yeh. A Prototyping Language for Real-Time Software. *IEEE Transactions on Software Engineering*, 14(10):1409 – 1423, October 1988.

[LWBL92] Gary T. Leavens, Tim Wahls, Albert L. Baker, and Kari Lyle. A Structural Operational Semantics of Firing Rules for Structured Analysis Style Data Flow Diagrams. Technical Report TR, Department of Computer Science, Iowa State University, Ames Iowa 50010, 1992.

[Lyl92] Kari Lyle. Refinement in Data Flow Diagrams. Master's thesis, Iowa State University, Ames, Iowa, 50011, 1992.

[Mey90] Bertrand Meyer. Lessons from the Design of the Eiffel Libraries. *Communications of the ACM*, 33(9):69–88, September 1990.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, Berlin, 1992.

[Nor90] N. D. North. An Implementation of Sets and Maps as Miranda Abstract Data Types. Technical Report DITC 162/90, National Physical Laboratory, Teddington, Middlesex TW11 OLW, United Kingdom, February 1990.

[O'N89] Guy O'Neill. Rapid Prototyping of Formal Specifications Using Miranda. Technical Report DITC 150/89, National Physical Laboratory, Teddington, Middlesex TW11 OLW, United Kingdom, November 1989.

[O'N92a] Guy O'Neill. Automatic Translation of VDM Specifications into Standard ML Programs. Technical Report DITC 196/92, National Physical Laboratory, Teddington, Middlesex TW11 OLW, United Kingdom, February 1992.

[O'N92b] Guy O'Neill. Automatic Translation of VDM Specifications into Standard ML Programs. *The Computer Journal*, 35(6):623–624, December 1992.

[Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.

[PJ86] Simon Peyton-Jones. Functional Programming Languages as Software Engineering Tools. In D. Ince, editor, *Software Engineering: The Decade of Change*, pages 124 – 153. Peter Peregrinus Ltd., London, UK, 1986.

[Pnu86] A Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends. In W. P. de Roever and G. Rozenberg, editors, *Current Trends in Concurrency: Overviews and Tutorials*, volume 224 of *Lecture Notes in Computer Science*, pages 510 – 584. Springer-Verlag, Berlin, 1986.

[Ram91] Norman Ramsey. Literate Programming Tools Need Not Be Complex. Technical Report CS-TR-351-91, Princeton University, 1991.

[RC93] G-H. B. Rafsanjani and S. J. Colwill. From Object-Z to C++: A Structural Mapping. In J. P. Bowen and J. E. Nicholls, editors, *Z User Work-*

shop, London 1992, Workshops in Computing, pages 166–179, Berlin, 1993. Springer-Verlag.

[RE93] M. Andersen R. Elmstrøm, P.B. Lassen. An Executable Subset of VDM-SL, in an SA/RT Framework. 1993. Submitted for publication in Real-Time Systems Journal.

[SBC92] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, Berlin, 1992.

[Sch86] David A. Schmidt. *Denotational Semantics — A Methodology for Language Development.* Wm. C. Brown Publishers, Dubuque, Iowa, 1986.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1988.

[Spi89] J. M. Spivey. An Introduction to Z and Formal Specifications. *Software Engineering Journal*, pages 40 - 50, January 1989.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, New York, second edition, 1992. ISBN 013983768X.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[TC89] R.B. Terwilliger and R.H. Campbell. PLEASE: Executable Specifications for Incremental Software Development. *Journal of Systems and Software*, 10(2):97 - 112, September 1989.

[Tur85] D. Turner. Miranda: A Non-strict Functional Language with Polymorphic Types. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science Vol. 201).* Springer Verlag, New York, 1985.

[TY92] S. Tyszberowicz and A. Yehudai. OBSERV - A Prototying Language and Environment. *ACM Transactions on Software Engineering and Methodology*, 1(3):269 - 309, July 1992.

[Van89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, Cambridge, Massachusetts, 1989.

[War86] Paul T. Ward. The Transformation Schema: an Extension of the Data Flow Diagram to Represent Control and Timing. *IEEE Transactions on Software Engineering*, (2):198 - 210, February 1986.

[WBL93] Tim Wahls, Albert L. Baker, and Gary T. Leavens. An Executable Semantics for a Formalized Data Flow Diagram Specification Language. Technical Report TR93-27, Department of Computer Science, Iowa State University, Ames, Iowa 50011, November 1993. available by anonymous ftp from ftp.cs.iastate.edu and by e-mail from almanac@cs.iastate.edu.

[WBL94] Tim Wahls, Albert L. Baker, and Gary T. Leavens. The Direct Execution of SPECS-C++: A Model-Based Specification Language for C++ Classes. Technical Report TR94-02, Department of Computer Science, Iowa State University, Ames, Iowa 50011, February 1994. available by anonymous ftp from ftp.cs.iastate.edu and by e-mail from almanac@cs.iastate.edu.

[WE92] M.M. West and B.M. Eaglestone. Software development: two approaches to animation of Z specifications using Prolog. *Software Engineering Journal*, 7(4):264–276, July 1992.

[Wil92] Alan Wills. Specification in Fresco. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[YB92] H. Yin and J.M. Bieman. A Run-time Type System for Monitoring Invariants. 1992. submitted to ACM Letters on Programming Languages and Systems.

[You89] Edward Yourdon. *Modern Structured Analysis.* Yourdon Press computing series. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[ZS86] P. Zave and W. Schell. Salient Features of an Executable Specification Language and Its Environment. *IEEE Transactions on Software Engineering*, 12(2):312 - 325, February 1986.